Algoritmos Evolutivos

Un enfoque práctico



Lourdes Araujo

Carlos Cervigón

Alfaomega Ra-Ma®

Algoritmos evolutivos: un enfoque práctico

Lourdes Araujo

Carlos Cervigón

Alfaomega Ra-Ma®

Datos catalográficos
Araujo, Lourdes y Cervigón, Carlos.
Algoritmos Evolutivos: un enfoque práctico.
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México
ISBN: 978-607-7686-29-3
Formato: 17 x 23 cm Páginas: 332

Algoritmos evolutivos: un enfoque práctico
Lourdes Araujo y Carlos Cervigón
ISBN: 978-84-7897-911-0, edición original publicada por RA-MA Editorial, Madrid, España

Primera edición: Alfaomega Grupo Editor, México, abril 2009

Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-7686-29-3

Impreso en México. Printed in Mexico.

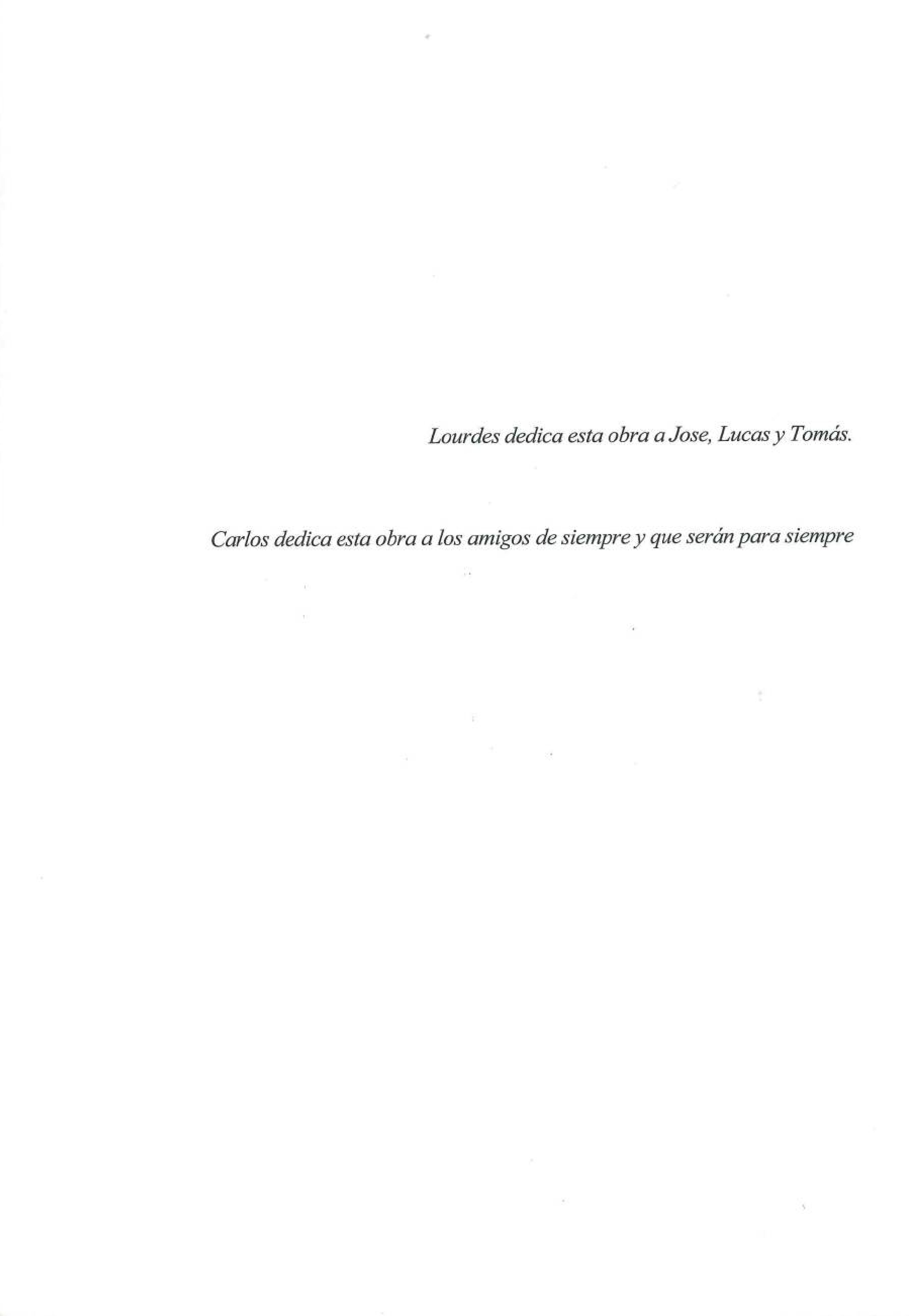
Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Pitágoras 1139, Col. Del Valle, México, D.F. – C.P. 03100.
Tel.: (52-55) 5089-7740 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Carrera 15 No. 64 A 29 – PBX (57-1) 2100122, Bogotá, Colombia, Fax: (57-1) 6068648 – E-mail: scliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – General del Canto 370-Providencia, Santiago, Chile
Tel.: (56-2) 235-4248 – Fax: (56-2) 235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Paraguay 1307 P.B. "11", Buenos Aires, Argentina, C.P. 1057 – Tel.: (54-11) 4811-7183 / 8352, E-mail: ventas@alfaomegaeditor.com.ar



Lourdes dedica esta obra a Jose, Lucas y Tomás.

Carlos dedica esta obra a los amigos de siempre y que serán para siempre

ÍNDICE

PRÓLOGO

Los **Algoritmos Evolutivos** constituyen una técnica general de resolución de problemas de búsqueda y optimización. Su forma de procesamiento se ha inspirado en la teoría de la evolución de las especies. Trabajan con una colección o "población" de soluciones candidatas o "individuos", para los que se calcula una medida de su "adaptación" o capacidad de ser solución al problema a resolver. La composición de la población va cambiando a lo largo de un proceso iterativo, cuyas iteraciones se denominan generaciones. Los individuos con mayor adaptación tienen una probabilidad mayor de sobrevivir y permanecer en la población de la siguiente generación, y de participar en "operaciones genéticas", que son operaciones de creación de nuevos individuos a partir de modificaciones de los de la población anterior. De esta forma, se emula el proceso de la selección natural.

Los mecanismos de creación de nuevos individuos a lo largo del proceso evolutivo aportan a estos algoritmos una de sus propiedades fundamentales: su capacidad de acceder a cualquier región del espacio de búsqueda del problema. Sin embargo, esta capacidad no conlleva la ineficiencia de una búsqueda aleatoria, ya que el mecanismo de selección introduce un sesgo hacia las regiones más prometedoras.

Los algoritmos evolutivos permiten abordar problemas complejos de búsqueda y optimización que surgen en las ingenierías y los campos científicos: problemas de planificación de tareas, horarios, tráfico aéreo y ferroviario, búsqueda de caminos óptimos, optimización de funciones, etc. Con este libro hemos querido aportar un enfoque práctico al estudio de estos algoritmos evolutivos, que es

fundamental para aplicarlos a problemas reales de cualquier disciplina del conocimiento.

Los algoritmos evolutivos presentan una estructura general que puede aplicarse a los distintos problemas, facilitando enormemente las tareas de diseño e implementación. El único requisito de un usuario que desee aplicar esta técnica para resolver un problema concreto es saber programar en cualquier lenguaje de propósito general en el que codificará el algoritmo evolutivo. Sin embargo, para obtener buenos resultados con estos algoritmos es necesario conocerlos con detalle, ya que dentro del esquema general de un algoritmo evolutivo hay que elegir múltiples componentes y parámetros, de los que va a depender la calidad del resultado y la eficiencia del algoritmo. El conocimiento de la elección más adecuada en cada caso, que a menudo depende de detalles sutiles del problema considerado, sólo se consigue con la práctica. Esta idea nos ha llevado a proponer este libro, que consideramos adecuado para cualquier ingeniero o licenciado con conocimientos básicos de programación.

Este libro ha partido de las notas de los cursos impartidos durante los últimos seis años por los autores en la asignatura de *Programación evolutiva*, en la Ingeniera Informática de la Universidad Complutense. El libro tiene dos partes: una primera en la que se describen los algoritmos, y una segunda en la que se proponen varios proyectos y se resuelven empleando estas técnicas.

En la primera parte se presentan los conceptos que se manejan en el diseño de los algoritmos evolutivos, siempre con un enfoque práctico orientado al desarrollo de aplicaciones. El primer capítulo de esta parte es una introducción al tema, sus antecedentes y sus variantes. El segundo capítulo está dedicado a uno de los tipos más frecuentemente utilizados de algoritmos evolutivos: los algoritmos genéticos. Estos algoritmos, que trabajan con una representación sencilla de los problemas en forma de cadenas binarias, nos sirven para presentar la estructura y componentes básicos de los algoritmos evolutivos. Para ello introducimos una sencilla notación en pseudocódigo en la que se formulan los algoritmos a lo largo del libro. En el tercer capítulo se describen distintos refinamientos del algoritmo básico presentado en el capítulo anterior. El cuarto capítulo está dedicado a otras variantes de algoritmos evolutivos, que trabajan con distintos tipos de representación de las soluciones del problema considerado. Finalmente, en el quinto capítulo se presentan alternativas que afectan a la estructura general del algoritmo, como los algoritmos evolutivos paralelos o los que buscan la optimización simultánea de varios objetivos (optimización multiobjetivo).

En la segunda parte del libro se describe con detalle una serie de proyectos prácticos de resolución de problemas complejos aplicando distintos tipos de

algoritmos evolutivos. El primero de estos proyectos es una aplicación directa del diseño presentado en la parte I, para facilitar al lector la comprensión y manejo de los algoritmos. Los restantes proyectos, muy variados, abordan distintos problemas de alta complejidad computacional, que los algoritmos evolutivos permiten abordar con facilidad: planificación de horarios, de recursos, búsqueda de caminos óptimos con diversas restricciones y búsquedas de estrategias de juego, por citar algunos de ellos. Cada uno de los proyectos se describe cuidadosamente, especificando los elementos del diseño del tipo de algoritmo evolutivo elegido para resolver el problema. Las propuestas que se presentan facilitarán a los lectores el acercamiento al problema y a otros similares, y a su resolución con un algoritmo evolutivo. Estas propuestas pueden refinarse aplicando variantes de las presentadas en la parte I. También se proponen para todas ellas versiones más complejas que el usuario puede desarrollar como ejercicios. Los esquemas se han planteado en pseudocódigo con la intención de que sean accesibles para personas que trabajen en cualquier lenguaje de programación. Al final del libro, en los apéndices, apuntamos algunos detalles prácticos para la programación en dos de los lenguajes más extendidos: C++ y Java.

Nos gustaría terminar este preámbulo con una mención especial a Darwin en el bicentenario de su nacimiento y en el 150 aniversario de la publicación del "*Origen de las especies*", por su genial aportación a la ciencia y su inspiración para la Programación Evolutiva.

Capítulo 1

AES: TÉCNICAS DE BÚSQUEDA Y OPTIMIZACIÓN

1. INTRODUCCIÓN

Los **Algoritmos Evolutivos** (AEs) son una técnica de resolución de problemas de búsqueda y optimización inspirada en la teoría de la evolución de las especies y la selección natural. Estos algoritmos reúnen características de búsqueda aleatoria con características de búsqueda dirigida que provienen del mecanismo de selección de los individuos más adaptados. La unión de ambas características les permite abordar los problemas de una forma muy particular, ya que tienen capacidad para acceder a cualquier región del espacio de búsqueda, capacidad de la que carecen otros métodos de búsqueda exhaustiva, a la vez que exploran el espacio de soluciones de una forma mucho más eficiente que los métodos puramente aleatorios. Indudablemente, un algoritmo diseñado de forma específica para la resolución de un problema concreto será más eficiente que un algoritmo evolutivo, que es una técnica general de resolución. Pero existen muchas situaciones en las que no es posible contar con tales algoritmos.

Los algoritmos evolutivos proporcionan un esquema general para la resolución de problemas. Es decir, tenemos el algoritmo diseñado para el problema que nos ocupa y sólo tenemos que especificar la forma de ciertos componentes. Incluso en el diseño de estos componentes, hay patrones de diseño que se pueden aplicar a toda una clase de problemas y que facilitan la construcción del algoritmo evolutivo. En general, la parte más dependiente del problema específico considerado es la definición de la función de adaptación, que se utiliza en el proceso de selección del algoritmo evolutivo. Tanto en el diseño de esta función

como en el resto de los componentes del algoritmo es fácil incorporar conocimiento específico del problema, si es que se dispone de él, lo que puede mejorar significativamente la eficiencia y la calidad de las soluciones encontradas.

También es importante tener en cuenta que los algoritmos evolutivos no garantizan una solución exacta al problema abordado, sino una aproximación cuya calidad dependerá de los recursos dedicados a la búsqueda, es decir, tiempo y memoria, aparte, claro está, del diseño adecuado de los componentes del algoritmo. Gracias a esta característica pueden aplicarse a problemas de alta complejidad computacional, proporcionando una solución aproximada que en muchos casos es suficiente para las necesidades del usuario, o en otros es lo único que se puede obtener.

Aunque la estructura general de los algoritmos evolutivos facilita enormemente el diseño de un algoritmo de resolución de un nuevo problema, sólo la práctica y la experiencia con este tipo de algoritmos nos permite sacarles el máximo partido, ya que sus resultados son muy dependientes de la correcta elección de sus componentes y parámetros. Por esto, en este libro se presenta una colección de problemas reales, y su resolución con un algoritmo evolutivo. El estudio y puesta en práctica de estos proyectos permitirá al lector conocer y obtener la experiencia necesaria con los algoritmos evolutivos para llegar a comprender su filosofía y para abordar el diseño del algoritmo que resuelva un nuevo problema.

1.1 LA TEORÍA DE LA EVOLUCIÓN

Los Algoritmos Evolutivos parten de las ideas del modelo de evolución natural que fue propuesto por Charles Darwin [DAR1859]. De acuerdo con la teoría de Darwin la evolución de las especies se debe al principio de *selección natural*, que favorece la supervivencia y multiplicación de aquellas especies que están mejor adaptadas a las condiciones de su entorno. Otro elemento que Darwin señaló como relevante para la evolución son las *mutaciones*, o pequeñas variaciones que introducen diferencias en las características físicas y tipos de respuesta de los padres y los hijos. El mecanismo que fuerza la actuación de la selección es la producción de descendencia. Mientras hay abundancia de recursos, la población crece exponencialmente. Este proceso lleva a situaciones de escasez de recursos en el entorno, en las que los individuos "mejor adaptados" al medio tienen mayor probabilidad de sobrevivir y de dejar descendencia.

La genética y las leyes de la herencia genética han complementado la teoría de Darwin con mecanismos relativos a la herencia de características físicas en la producción de descendencia, dando lugar a la teoría del **neodarwinismo**. De acuerdo con esta teoría, las características físicas de un individuo, su *fenotipo,* son

la consecuencia de su información genética o *genotipo*, cadenas de *genes* con complejas interacciones, que constituyen las unidades de transferencia de la herencia. Los genes pueden modificarse puntualmente por *mutaciones*. La replicación de las cadenas de genes en la reproducción no siempre es perfecta. A veces, aunque con una frecuencia extremadamente baja, se producen errores en el proceso de copia que constituyen mutaciones. Sin embargo, existen mecanismos reparadores de estos errores, como enzimas codificadas en los propios genes, que reducen aún más el porcentaje de este tipo de mutaciones. También existen factores externos, como la radiación y ciertas sustancias químicas, que pueden incrementar de forma significativa las probabilidades de mutación.

La selección tiene lugar sobre los individuos, que son la consecuencia del genotipo y su interacción con el medio, constituyendo las *unidades de selección*. Lo que evoluciona es el conjunto de individuos que constituyen la *población*, que representa a un conjunto de genes comunes a sus individuos. La *adaptación* de su individuo es su tendencia, relativa al resto de los individuos de la población, para sobrevivir y dejar descendencia en unas condiciones ambientales específicas.

Estas ideas están en la base del diseño de los algoritmos evolutivos. Además, muchas de las propiedades de la evolución de los seres vivos, como la edad, la mayor o menor tendencia a la mutación según el estadio de la evolución, etc., están siendo objeto de investigación para su incorporación a las técnicas de computación evolutiva. Sin embargo, los algoritmos evolutivos no tratan de ser un reflejo fiel de la evolución biológica. Debemos tener en cuenta que la naturaleza evoluciona a lo largo de millones de años, mientras que a nosotros nos interesa que nuestros algoritmos nos proporcionen una solución en un tiempo algo más corto.

1.2 ESQUEMA GENERAL DE UN ALGORITMO EVOLUTIVO

Los distintos algoritmos evolutivos que se pueden formular responden a un esquema básico común, y comparten una serie de propiedades:

- Procesan simultáneamente, no una solución al problema, sino todo un conjunto de ellas. Estos algoritmos trabajan con alguna forma de representación de soluciones potenciales al problema, que se denominan *individuos*. El conjunto de todos ellos forma la *población* con la que trabaja el algoritmo.

- La composición de la población se va modificando a lo largo de las iteraciones del algoritmo que se denominan generaciones. De generación

en generación, además de variar el número de copias de un mismo individuo en la población, también pueden aparecer nuevos individuos generados mediante operaciones de transformación sobre individuos de la población anterior. Dichas operaciones se conocen como operadores genéticos.

- Cada generación incluye un proceso de selección, que da mayor probabilidad de permanecer en la población y participar en las operaciones de reproducción a los mejores individuos. Los mejores individuos son aquellos que dan lugar a los mejores valores (ya sean mínimos o máximos) de la función de adaptación del algoritmo. Es fundamental para el funcionamiento de un algoritmo evolutivo que este proceso de selección tenga una componente aleatoria, de forma que individuos con baja adaptación también tengan oportunidades de sobrevivir, aunque su probabilidad sea menor. Es esta componente aleatoria la que dota a los algoritmos evolutivos de capacidad para escapar de óptimos locales y de explorar distintas zonas del espacio de búsqueda.

El siguiente pseudocódigo muestra un posible esquema general de un algoritmo evolutivo:

```
funcion Algoritmo_Genético()
{
     TPoblacion pob;     // población
     TParametros parámetros// tamaño población

     obtener_parametros(parametros);
     pob = poblacion_inicial();
     evaluacion(pob, tam_pob, pos_mejor, sumadaptacion);

     // bucle de evolución
     mientras no se alcanza condición de terminación hacer{
        seleccion(pob, parámetros);
        reproduccion(pob, parámetros);
        evaluacion(pob,parámetros,pos_mejor,sumadaptacion);
      }
     devolver pob[pos_mejor]
}
```

El algoritmo procesa un conjunto de individuos que forman la población *pob*. Al comienzo del algoritmo se obtienen los datos de entrada al problema (*obtener_parámetros*) y se genera la población inicial, cuyos individuos se evalúan mediante la función de adaptación del algoritmo. El resto del algoritmo consiste en un bucle, cada una de cuyas iteraciones es una generación en la que se produce un

proceso de selección, que da mayores probabilidades de tener copias en la nueva población a los individuos más adaptados, seguido de un proceso de reproducción en el que se generan nuevos individuos a partir de los de la población mediante operaciones de mezcla y pequeñas alteraciones, y finalmente una evaluación de la nueva población. En muchas ocasiones se utilizan pequeñas variantes de este esquema. Así, por ejemplo, a veces se selecciona un subconjunto de la población, que es el único que participa en las operaciones de reproducción.

Siguiendo este esquema general se han desarrollado distintas variantes de algoritmos evolutivos, cuya principal diferencia se encuentra en la representación de los individuos. Lógicamente, los operadores genéticos que se utilizan para la reproducción en cada caso dependen de la representación adoptada. Las variantes más conocidas, y de las que nos ocupamos en este libro, son:

- *Los Algoritmos Genéticos* [HOL75]: utilizan una representación binaria o entera.

- *Los Programas de Evolución* [MIC94]: los individuos son cualquier estructura de datos de tamaño fijo.

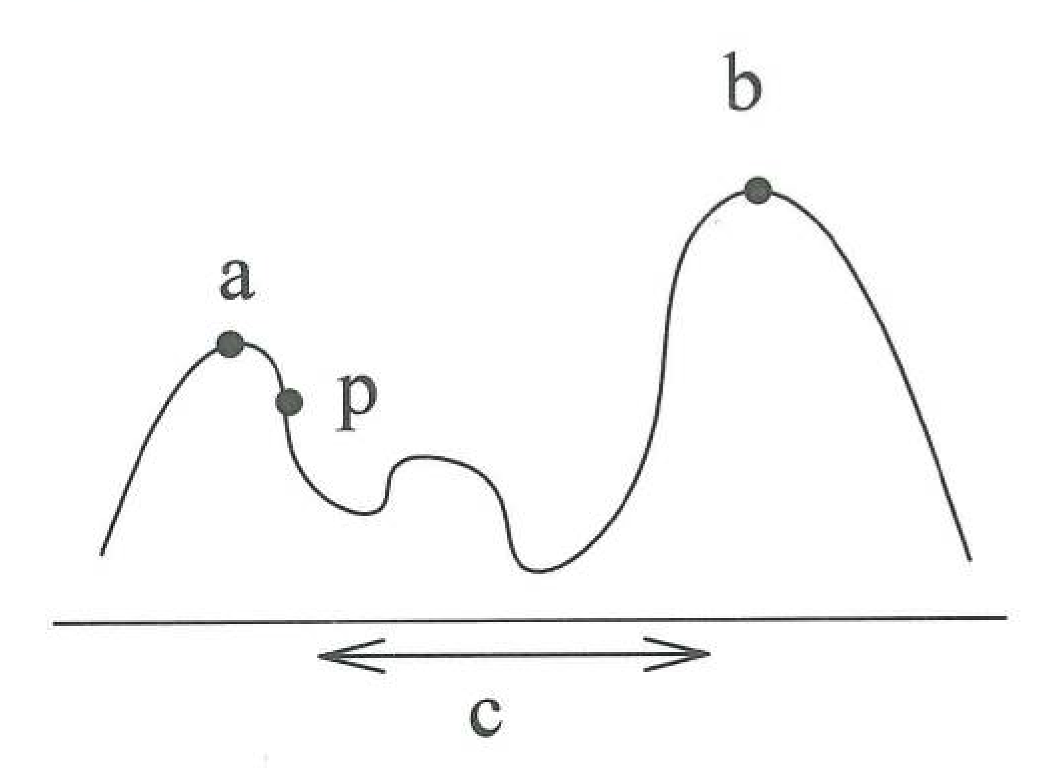
- *Algoritmos Genéticos de Codificación Real* [BAC91, GOL91, HER98]: se hace evolucionar una población de números reales que codifican las posibles soluciones de un problema numérico.

- *La Programación Evolutiva* [KOZ94, KOZ02]: se hace evolucionar una población de "programas" para resolver un problema en general. Los programas que evolucionan pueden adoptar distintas formas, aunque la más usual es un árbol. En cualquier caso, se trata de estructuras de datos de tamaño variable, es decir, no todos los individuos tienen el mismo tamaño. Este tipo de AE presenta una diferencia fundamental con el resto: no buscan la solución a una instancia concreta de un problema, sino una estrategia capaz de resolver cualquier instancia de ese problema.

1.3 BÚSQUEDA Y OPTIMIZACIÓN

Gran parte de los problemas que surgen en el desarrollo industrial y en la investigación pueden formularse como una búsqueda o como una optimización: dado un sistema, buscamos un conjunto de valores que permiten llevarlo a una determinada configuración o bien un conjunto de valores que permiten optimizar su comportamiento (rendimiento, calidad, coste, etc.).

El método clásico de optimización para problemas cuyo espacio de búsqueda de soluciones es continuo es la técnica de *escalada*, que consiste en determinar la pendiente de la vecindad del punto actual y seleccionar el punto de mayor pendiente en dicha vecindad. Si el valor de la función a optimizar en el nuevo punto es mejor que en el anterior, el nuevo punto se convierte en el punto actual. El proceso continúa hasta que no es posible realizar ninguna mejora. Una limitación de este método es su incapacidad para escapar de óptimos locales. Consideremos una función de una variable como la de la figura:



Si la exploración comienza en un punto como *p*, sólo será capaz de llegar al máximo *a*, pero no podrá llegar a *b*, ya que para ello tendrá que atravesar la región *c* de valores peores que *a*.

Aparte de los problemas mencionados de los métodos clásicos, otra dificultad para resolver problemas de optimización está en su complejidad computacional. La mayor parte de los problemas de optimización que surgen en el mundo real tienen una complejidad exponencial, es decir, no existe un procedimiento para resolverlos que opere en un número de pasos que sea una función polinómica del tamaño del problema. Esto implica que los métodos deterministas de resolución, como el de escalada, no son capaces de resolverlos a partir de cierto tamaño del problema.

Una alternativa es realizar una búsqueda aleatoria, tomando puntos al azar dentro de una zona de búsqueda, y estimando el valor del óptimo mediante técnicas estadísticas. Estas técnicas se pueden aplicar a cualquier tipo de problema, pero son muy ineficientes.

Los algoritmos evolutivos constituyen un punto intermedio al reunir componentes de búsqueda aleatoria y de búsqueda dirigida por la selección. Una representación muy extendida de la evolución de soluciones son las *superficies adaptativas* [BAC96A]. Las posibles combinaciones de rasgos físicos en una población de individuos definen puntos en un espacio multidimensional, donde cada eje de coordenadas corresponde a uno de estos rasgos. Esta representación usa una dimensión adicional para dibujar los valores de adaptación de cada punto del espacio, reflejando la ventaja selectiva (o desventaja) de los individuos correspondientes. De esta forma se construye la *superficie adaptativa*, que en su forma simplificada tridimensional (dos dimensiones de rasgos, y una de adaptación) tiene el aspecto de una zona de valles y picos. Cuando las combinaciones de rasgos de la población cambian, y la adaptación media se incrementa, la población se mueve hacia arriba, y de esta forma, escala alguno de los picos. Así la evolución se puede interpretar como un proceso de optimización de la adaptación análogo al que necesitamos realizar para resolver un problema de optimización. Sin embargo, los algoritmos evolutivos no escalan de forma uniforme hacia regiones de mejor adaptación, sino que cuentan con mecanismos que les permiten cruzar regiones de baja adaptación. Estos mecanismos son fundamentalmente la selección no determinista y las mutaciones.

Capítulo 2

ALGORITMOS GENÉTICOS

Uno de los tipos de algoritmos evolutivos más populares son los algoritmos genéticos (AGs), propuestos por Holland [HOL75] en 1975. Se caracterizan por representar las soluciones al problema que abordan en forma de cadena de bits. Entre las razones que hacen que este tipo de AE suela ser uno de los estudiados con más detalle están su eficiencia y sencillez de implementación. Otra razón importante es la existencia de numerosos estudios teóricos de los mecanismos subyacentes al funcionamiento de estos algoritmos para la resolución de problemas.

Empieza a ser tradicional [GOL89, MIC94] comenzar los libros de algoritmos evolutivos con un ejemplo de aplicación a algún problema de planteamiento claro. Este texto también sigue esta línea, ya que la mejor forma de realizar un acercamiento práctico a estos algoritmos es resolver un problema aplicando un AE configurado con una selección sencilla de características. Así pues, comenzaremos presentando los elementos que se requieren para aplicar el esquema general, considerando para cada uno de ellos sus formas más tradicionales o más sencillas. Otras variantes más complejas o refinadas se irán presentando a lo largo del libro. Una vez detalladas las características del algoritmo, las aplicaremos a un problema sencillo: la optimización de funciones. Este tipo de problemas se suele utilizar tanto en la enseñanza de los AEs como en las evaluaciones de nuevos AEs que se investigan, tanto por su sencillez como por la posibilidad de configurar su grado de dificultad: número de óptimos de la función y número de variables implicadas.

2.1 PRINCIPALES ELEMENTOS DE UN AG

Los AGs, así como otras clases de AEs, responden en general a un mismo esquema, que representábamos en el capítulo anterior, y que aquí reproducimos:

```
funcion Algoritmo_Genético()
{
 TPoblacion pob;     // población
 TParametros parámetros// tamaño población

 obtener_parametros(parametros);
 pob = poblacion_inicial();
 evaluacion(pob, tam_pob, pos_mejor, sumadaptacion);
 // bucle de evolución
 mientras no se alcanza la condición de terminación hacer{
      selección(pob, parámetros);
      reproduccion(pob, parámetros);
      evaluacion(pob, parámetros, pos_mejor, sumadaptacion);
 }
devolver pob[pos_mejor];
}
```

La ejecución de un AE requiere una serie de parámetros de funcionamiento, como por ejemplo el tamaño de la población con la que va a trabajar, que definen su comportamiento en promedio. Una vez que el algoritmo dispone de los valores para estos parámetros, comienza generando una población de individuos, cada uno de los cuales es un candidato a ser solución del problema tratado, o permite llegar a la solución a partir de él. A continuación la población de individuos se somete a un bucle de evolución cada uno de cuyos ciclos incluye un proceso de selección, que modifica la composición de la población, eliminando a ciertos individuos y reforzando la presencia de otros, a un proceso de reproducción, que introduce nuevos individuos, y una nueva evaluación, que actualiza los datos de evolución, tales como la adaptación media de la población o la posición del mejor individuo de la población.

Como apuntábamos en el capítulo anterior, este esquema general da lugar a diversas clases de AEs en función de la representación adoptada para los individuos que componen la población. Y dentro de cada una de esas clases, cada algoritmo es un caso particular, no sólo en función del problema concreto al que se aplica, sino también dependiendo de la elección concreta de los métodos que se aplican en los distintos procesos involucrados, como la selección y la reproducción.

A continuación vamos a detallar los elementos del esquema general para el caso de los algoritmos genéticos. Para cada uno de estos elementos veremos algunos de las variantes más básicas o más usuales. Los procedimientos que se

presentan que se refieren a la representación de los individuos son específicos de los algoritmos genéticos. Sin embargo, otros son aplicables a cualquiera de las clases de algoritmos evolutivos.

2.2 REPRESENTACIÓN DE LOS INDIVIDUOS

En un AG los individuos son cadenas binarias, que denotaremos por **b**, que representan a puntos **x** del espacio de búsqueda del problema. Tomando la nomenclatura de la biología, a **b** se le denomina *genotipo* del individuo y a **x** se le denomina *fenotipo*. La nomenclatura biológica a veces se adopta también para otros datos del individuo. Así se usa *gen* para referirse a la codificación de una determinada característica del individuo. En los AGs se suele identificar un gen con cada posición de la cadena binaria, aunque esto no tiene por qué ser siempre así. Se usa *alelo* para los distintos valores que puede tomar un gen y *locus* para referirse a una determinada posición de la cadena binaria.

La sencillez de la representación binaria que utilizan los AGs les aporta características muy importantes de eficiencia que veremos en la sección 2.11. La contrapartida es que es necesario disponer de un método para pasar de la representación binaria al espacio de búsqueda natural al problema. Este paso es necesario en general para poder evaluar la adecuación del individuo como solución al problema. Lógicamente, el método de transformación es específico del problema considerado. Sin embargo, a la hora de diseñar el método de codificación es importante tener en cuenta una serie de directrices. Así, debemos buscar una *codificación* tal que cada punto del espacio de búsqueda esté representado por el mismo número de cadenas binarias, y tal que sea capaz de representar todos los puntos del espacio del problema.

Es importante que cada posición de la cadena tenga un significado para el problema, ya que de esta forma se favorece que los genes que dan alta calidad a un individuo sigan dando lugar a características de calidad en un nuevo individuo obtenido por alguna operación genética a partir del primero. También es importante buscar una codificación a partir de la cual se pueda llegar de forma eficiente al fenotipo, ya que la decodificación será una operación muy frecuente a lo largo de la evolución.

2.3 GENERACIÓN DE LA POBLACIÓN INICIAL

Los individuos de la población inicial de un AG suelen ser cadenas de ceros y unos generadas de forma completamente aleatoria, es decir, se va generando cada gen con una función que devuelve un cero o un uno con igual

probabilidad. En algunos problemas en los que se disponga de información adicional que nos permita saber de antemano que determinadas cadenas tienen más probabilidades de llegar a ser solución, podemos favorecer su generación al crear la población inicial. Sin embargo, es imprescindible para el buen funcionamiento del AG dotar a la población de suficiente variedad para poder explorar todas las zonas del espacio de búsqueda.

2.4 GRADO DE ADAPTACIÓN DE LOS INDIVIDUOS

La evolución de la población depende de la calidad relativa de los individuos que compiten por aumentar su presencia en la población y por participar en las operaciones de reproducción. En un problema de búsqueda u optimización, dicha calidad se mide por la adecuación o *adaptación* de cada individuo a ser solución al problema. Es frecuente que los problemas se presenten como la optimización de una función matemática explícita. En dichos casos la función de adaptación coincide con la función a optimizar.

Sin embargo, a veces se realizan algunas transformaciones a la función a optimizar o función de *evaluación* $g(x)$ para transformarla en una función de adaptación adecuada $f(x)$. Denominaremos adaptación bruta de un individuo x a $g(x)$, y simplemente adaptación a $f(x)$.

2.5 CONDICIONES DE TERMINACIÓN

Es necesario especificar las condiciones en las que el algoritmo deja de evolucionar y se presenta la mejor solución encontrada. La condición de terminación más sencilla es alcanzar un determinado número de generaciones de evolución. Otras condiciones, que a veces se utilizan de forma combinada, son alcanzar una solución de una determinada calidad o detectar que la mayor parte de la población ha convergido a una forma similar, careciendo de la suficiente diversidad para que tenga sentido continuar con la evolución.

2.6 EL PROCESO DE SELECCIÓN: MECANISMOS DE MUESTREO

La población del algoritmo genético se somete a un proceso de selección que debe tender a favorecer la cantidad de copias de los individuos más adaptados. Este proceso se puede realizar de formas muy diferentes:

Selección proporcional o por ruleta

La probabilidad de selección p_i de un individuo i con este método es proporcional a su adaptación relativa:

$$p_i = \frac{f(i)}{\overline{f}}$$

siendo $\overline{f}$ la adaptación media de la población.

Necesitamos generar un número aleatorio de acuerdo con la distribución de probabilidad dada por los p_i. Si contamos con un generador de números aleatorios que genera números de forma uniformemente distribuida a lo largo de un intervalo, como [0,1], podemos seguir el siguiente procedimiento [PER96]:

- Definimos las *puntuaciones acumuladas* de la siguiente forma:

$$\begin{aligned} q_0 &:= 0 \\ q_i &:= p_1 + \ldots + p_i \qquad (\forall i = 1, \ldots, n) \end{aligned}$$

- Se genera un número aleatorio $a \in [0,1]$.

- Se selecciona al individuo i que cumpla:

$$q_{i-1} < a < q_i$$

Este proceso se repite para cada individuo que se desee seleccionar.

Muestreo estocástico universal

Es un procedimiento similar al de muestreo por ruleta, pero en este caso se genera un sólo número aleatorio, y a partir de él se generan los k números que se necesitan (para generar k individuos) espaciados de igual forma. Los números se calculan de la siguiente forma:

$$a_j := \frac{a + j - 1}{k} \qquad (\forall j = 1, \ldots, k)$$

Una vez generados estos números, el método funciona de la misma forma que la selección por ruleta. Este método es más eficiente que el de la ruleta.

Selección por torneo

En la selección por torneo se elige aleatoriamente una pequeña muestra de la población y de ella se selecciona el individuo de mejor valor de adaptación. Normalmente se utiliza un tamaño pequeño para la muestra, de 2 ó 3 individuos. El proceso se repite hasta completar el número de individuos que se desee seleccionar. Esta selección puede realizarse de forma determinista o probabilística.

- En la selección por torneo determinista se selecciona aleatoriamente un número p de individuos. Habitualmente p toma el valor 2. Después, de entre los individuos seleccionados se elige el mejor, es decir, el de mayor valor de adaptación.

- El caso probabilístico es similar, excepto porque, una vez seleccionados los p individuos, en lugar de escoger siempre al mejor, la selección se hace con una cierta probabilidad. Si un número, que se genera aleatoriamente entre 0 y 1, es mayor que un cierto umbral especificado como un parámetro del algoritmo, entonces se elige al mejor, en otro caso al peor.

Muestreo por restos

Este método realiza una selección proporcional a la adaptación de los individuos, garantizando copias de los mejores individuos sin intervención del azar, dejando que el azar sólo intervenga en la parte de los individuos a seleccionar para completar la muestra. Concretamente, de cada individuo x_i se seleccionan $p_i.k$ copias para la muestra, siendo k el número de individuos a seleccionar, y pi la probabilidad de selección del individuo i. Los individuos que quedan hasta completar el tamaño k de la muestra se seleccionan por alguno de los métodos anteriores.

Existen otros mecanismos de muestreo poco usuales en los algoritmos genéticos como son la selección directa de los mejores individuos, o la selección completamente aleatoria. Estos métodos carecen o bien de la componente aleatoria o bien de la componente dirigida, que caracterizan a los algoritmos evolutivos.

Consideremos un ejemplo de funcionamiento de los métodos anteriores. Supongamos que tenemos la siguiente población de 8 individuos:

Individuo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Adaptación	4	1	1	2	3	2	5	2
p_i	0.2	0.05	0.05	0.1	0.15	0.1	0.25	0.1
q_i	0.2	0.25	0.3	0.4	0.55	0.65	0.9	1.0

En la tabla anterior aparecen las probabilidades pi de los individuos, proporcionales a su adaptación, y las puntuaciones acumuladas qi, calculadas como se ha descrito anteriormente en el método de selección por ruleta. q_0, que no aparece en la tabla, es por definición 0.

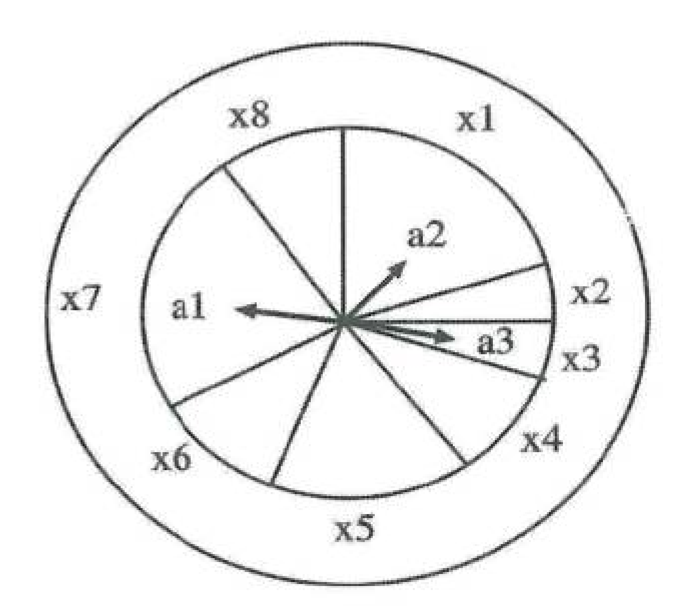
Supongamos que deseamos seleccionar 3 individuos por el método de la ruleta. Para ello generamos tres números aleatorios:

$$a_1 = 0.8751$$
$$a_2 = 0.1391$$
$$a_3 = 0.2582$$

De acuerdo con el método de la ruleta, los individuos seleccionados serán x_7, x_1 y x_3, ya que se cumplen las siguientes condiciones:

$$q_6 < a_1(0.8751) < q_7 \quad \rightarrow x_7$$
$$q_0 < a_2(0.1391) < q_1 \quad \rightarrow x_1$$
$$q_2 < a_3(0.2582) < q_3 \quad \rightarrow x_3$$

Si lo representamos gráficamente, la situación es la que muestra la siguiente figura:



Supongamos ahora que el método de selección que aplicamos es el muestreo estocástico universal. En este caso generamos un único número aleatorio a que suponemos que toma el siguiente valor:

$$a = 0.4578$$

Aplicando la fórmula de cálculo de los a_i por este método, con k=3 tenemos:

$$a_1 = \frac{0.4578 + 1 - 1}{3} = 0.1526$$
$$a_2 = \frac{0.4578 + 2 - 1}{3} = 0.48593$$
$$a_3 = \frac{0.4578 + 3 - 1}{3} = 0.81926$$

con lo que los individuos seleccionados en este caso serían x_1, x_6 y x_7.

$$q_0 < a_1(0.1526) < q_1 \quad \rightarrow x_1$$
$$q_4 < a_2(0.48593) < q_5 \quad \rightarrow x_5$$
$$q_6 < a_3(0.81926) < q_7 \quad \rightarrow x_7$$

Consideremos ahora el método de selección por torneos. Supongamos que el tamaño de la muestra es t=3. En esta ocasión generamos números aleatorios enteros e_i entre 1 y 8, que indican el individuo seleccionado.

Individuo	*Adaptación*
$e_1 = 5$	$\rightarrow 3$
$e_2 = 3$	$\rightarrow 1$
$e_3 = 7$	$\rightarrow 5$

Comparando los valores de las adaptaciones de los individuos seleccionados para la muestra, vemos que el mejor, suponiendo un problema de maximización, es el individuo 7 con adaptación 5. Por lo tanto, x1=7. Aplicando exactamente el mismo procedimiento se eligen x2 y x3.

Finalmente, si el método elegido es el muestreo por restos, calculamos el número de copias que le corresponde a cada individuo en la muestra de k=5 individuos a seleccionar, de acuerdo con su valor de adaptación:

Individuo	p_i	**Copias ($p_i.k$)**
1	0.2	1
2	0.05	0
3	0.05	0
4	0.1	0
5	0.15	0
6	0.1	0
7	0.25	1
8	0.1	0

Se seleccionan los individuos x_1 y x_7 y los tres que faltan para completar la muestra se seleccionan por alguno de los métodos anteriores, como por ejemplo la ruleta. Debemos observar que si hubiéramos tomado $k=3$ no se hubiera seleccionado directamente ninguno de los individuos de la población.

2.7 EL PROCESO DE REPRODUCCIÓN: OPERADORES GENÉTICOS

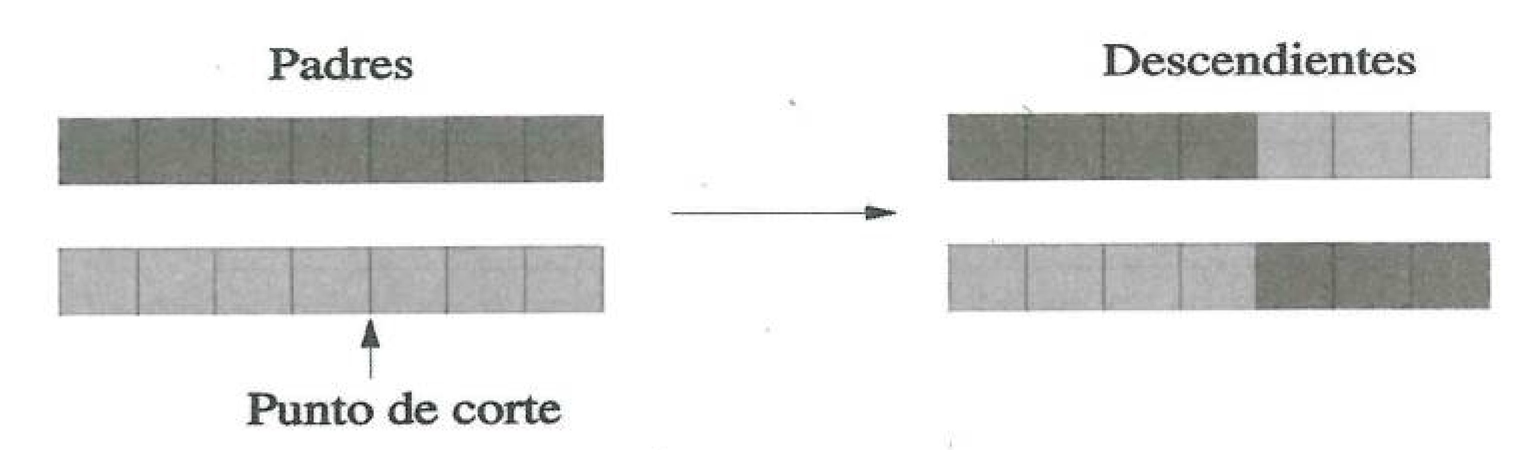
En cada nueva generación se crean algunos individuos que no estaban presentes en la población anterior. De esta forma el algoritmo genético va

accediendo a nuevas regiones del espacio de búsqueda. Los nuevos individuos se crean aplicando ciertos "operadores genéticos" a individuos de la población anterior. Los operadores que suelen estar presentes en todo algoritmo genético son los operadores de cruce y mutación. El operador de cruce combina propiedades de dos individuos de la población anterior para crear nuevos individuos. El operador de mutación crea un nuevo individuo realizando algún tipo de alteración, usualmente pequeña, en un individuo de la población anterior. Estos operadores pueden adoptar formas muy distintas. Veremos aquí la forma más simple de estos operadores y en otro capítulo distintas variantes que pueden adoptar.

Para cada operador genético se establece una tasa o frecuencia, de manera que el operador sólo se aplica si un valor generado aleatoriamente está por encima de la tasa especificada. Por ejemplo, si la tasa de aplicación del operador de cruce es del 40%, entonces si al generar un número aleatorio entre 0 y 1 obtenemos 0.56, que es mayor que 0.4, aplicamos el operador de cruce a los dos individuos seleccionados a tal efecto.

2.7.1 Operador de Cruce Monopunto

La forma más simple del operador de cruce en los algoritmos genéticos es el cruce monopunto. Este consiste en seleccionar al azar una única posición en la cadena de ambos padres e intercambiar las partes de los padres divididas por dicha posición, como muestra la siguiente figura:

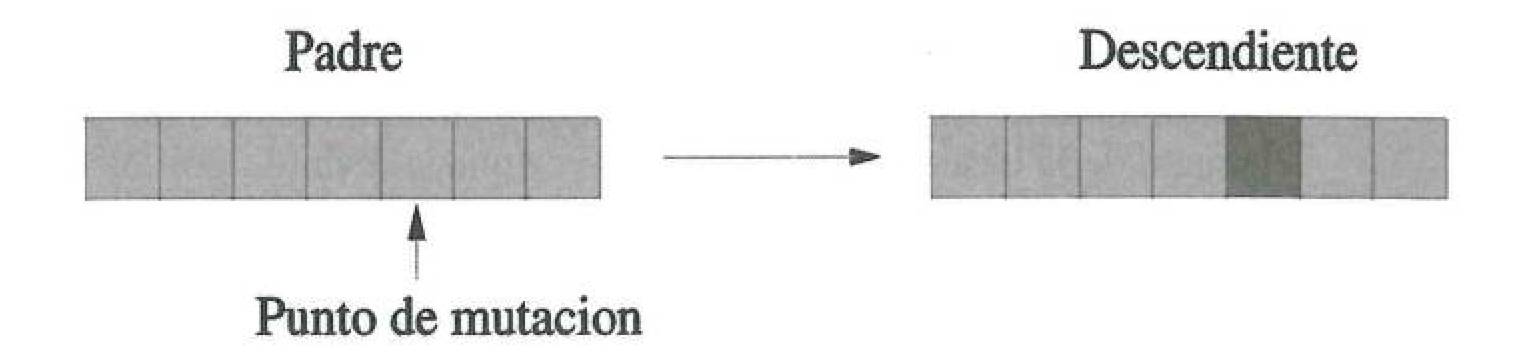


Este operador produce dos hijos que combinan propiedades de ambos padres, lo que puede llevar a una mejora de la adaptación de los hijos respecto a la de los padres.

2.7.2 Operador de Mutación Aleatoria bit a bit

La forma más sencilla de mutación que se puede aplicar a un individuo de un algoritmo genético consiste en cambiar el valor de una de las posiciones de la

cadena: si es cero pasa a uno, y si es uno pasa a cero. La siguiente figura muestra un ejemplo:



Para cada posición de cada individuo comprobamos la tasa de mutación, y si se cumple aplicamos el operador. Habitualmente la tasa de aplicación del operador de mutación es bastante pequeña (en torno al 0.1%) comparada con el operador de cruce. Sin embargo esto no es una norma general.

En muchos casos la mutación produce individuos con peor adaptación que los individuos originales, ya que la mutación puede romper las posibles correlaciones entre genes que se hayan formado con la evolución de la población. Sin embargo, contribuyen a mantener la diversidad de la población, que es fundamental para el buen funcionamiento del algoritmo.

2.8 EL PROCESO DE REEMPLAZO

Habitualmente los AGs mantienen un tamaño de población constante, aunque existen otras posibilidades. Para mantener el tamaño de la población, los nuevos individuos creados mediante los operadores genéticos deben reemplazar a otros de la población anterior. En función de la cantidad de individuos reemplazados en la población anterior se consideran distintos tipos de AGs:

- *AGS generacionales*: en estos algoritmos la población se renueva por completo de una generación a otra.

- *AGs con estado estacionario*: la descendencia de los individuos seleccionados en cada generación se incluye en la población, reemplazando a algunos de los individuos de la población anterior. En este caso se conserva parte de la población de generación en generación.

Existen diversas posibilidades para establecer los criterios de reemplazo en los AGs con estado estacionario. Los más usuales son los siguientes:

- *Reemplazo de los padres*: los hijos sustituyen a sus padres.
- *Reemplazo aleatorio*: los individuos a eliminar se eligen aleatoriamente. El número de individuos a eliminar viene dado por el tamaño de la descendencia, que a su vez queda definido por las tasas de cruces y mutaciones y el tamaño de la población.
- *Reemplazo de los individuos peor adaptados*: los individuos a eliminar se eligen aleatoriamente, pero sólo entre los que tiene el valor de adaptación más bajo. Valores bajos de adaptación se suelen considerar por debajo del 10% de la adaptación media.
- *Reemplazo de individuos de adaptación similar*: cada nuevo individuo reemplaza a un individuo de la población anterior que tiene un valor de adaptación similar al suyo. Para implementar este mecanismo se ordena a la población de acuerdo con su adaptación y se escoge para la sustitución una posición aleatoria de la "vecindad" de la posición que le corresponde al nuevo individuo. El tamaño de la vecindad depende del tamaño de la población. Son usuales valores entre 3 y 5.

2.9 IMPLEMENTACIÓN DEL ALGORITMO GENÉTICO SIMPLE

Goldberg [GOL89] propuso el denominado *algoritmo genético simple*, que utilizó para explicar con claridad el funcionamiento de un AG. Este algoritmo aplica en cada punto de diseño una de las elecciones más sencillas posibles en la implementación de un AG. Concretamente, las cadenas binarias de la población inicial se generan aleatoriamente, la función de adaptación es una función matemática para la que se busca el valor óptimo en un determinado intervalo, se aplica cruce monopunto y mutación aleatoria bit a bit. La selección se realiza por ruleta y se aplica reemplazo de los padres. Los individuos a cruzar se eligen de forma consecutiva, ya que se supone que el proceso de selección ha reubicado a los individuos de forma que un emparejamiento consecutivo de los individuos se pueda considerar aleatorio. El criterio de terminación es un número máximo de generaciones.

De acuerdo con las descripciones realizadas hasta el momento en este capítulo, podemos escribir de forma más detallada el esquema general de este AG como sigue:

```
funcion Alg_Gen ()
{
  TPoblacion pob;          // población
  entero tam_pob;          // tamaño población
  entero lcrom;            // tamaño población
  entero pos_mejor;        // posición del mejor cromosoma
  real sumadaptacion;      // adaptación de toda la población
  real prob_cruce;         // probabilidad de cruce
  real prob_mut;           // probabilidad de mutación

  obtener_parametros(tam_pob, lcrom);
  pob = poblacion_inicial(tam_pob, lcrom);
  evaluacion(pob, tam_pob, pos_mejor, sumadaptacion);

  // bucle de evolución
  para cada generacion desde 0 hasta num_max_gen hacer{
     selección(pob, parámetros);
     reproduccion(pob, tam_pob, lcrom, prob_cruce);
     mutacion(pob, tam_pob, lcrom, prob_mut);
     evaluacion(pob, tam_pob, pos_mejor, sumadaptacion);
  }
  devolver pob[pos_mejor];
}
```

Consideremos una posible implementación en pseudocódigo de este algoritmo genético. Detallaremos la forma de los operadores de reproducción, cruce y mutación aplicados a la optimización de una función simple de una variable codificada como un entero binario sin signo.

La Tabla 1 muestra algunas de las abreviaturas utilizadas en el pseudocódigo de los algoritmos. El pseudocódigo que se presenta aquí no pretende entrar en detalles de implementación como la elección de programación estructurada u orientada a objetos, o el uso de representaciones de memoria dinámica o estática que se necesiten en los algoritmos.

Por ello, aunque en algunos casos el código pueda sugerir una u otra elección, no hay ninguna necesidad de adherirse a ella en la implementación específica que se realice.

eoc	en otro caso
var	paso por variable
vector	representación estática o dinámica de una colección de variables del mismo tipo
abs	valor absoluto
pow	potencia
alea	devuelve un número aleatorio real entre 0 y 1
alea_ent(min,max)	devuelve un entero aleatorio entre *min* y *max*

Tabla 1: *Notación utilizada en el pseudocódigo de los algoritmos*

2.9.1 Estructuras de Datos

En primer lugar, necesitamos una representación del genotipo, es decir, de la cadena de genes.

```
tipo TGenes: vector de booleano;
```

Cada individuo incluye información sobre su genotipo y fenotipo (el valor real que representa) y otros datos necesarios para su evaluación, como su valor de adaptación, su puntuación relativa respecto a los restantes individuos y su puntuación acumulada, usada en los sorteos.

```
tipo TIndividuo = registro{
   TGenes genes;      // cadena de bits (genotipo)
   real x;            // fenotipo
   real adaptación;   // función de evaluación
   real puntuacion;   //punt.relativa:adaptación/sumadaptacion
   real punt_acu;     // puntuación acumulada para sorteos
}
```

Finalmente, representamos la población como una colección de individuos.

```
tipo TPoblacion: vector de TIndividuo;
```

2.9.2 Generación de la Población Inicial

Tras la inicialización de los parámetros, el primer paso del algoritmo es la inicialización de una población de **tam_pob** individuos.

```
TPoblacion poblacion_inicial(entero tam_pob,  entero lcrom)
{
  TPoblacion pob;
  TIndividuo indiv;
  entero i;
```

```
    para cada i desde 0 hasta tam_pob {
        indiv = genera_indiv(lcrom);
        pob[i] = indiv;
    }
    devolver pob;
}
```

La creación de cada individuo consiste en generar su genotipo como una cadena aleatoria de ceros y unos, y en calcular su adaptación.

```
TIndividuo genera_indiv(entero lcrom)
{
   entero i;
   TIndividuo indiv;
   entero gen;
   inicializar(indiv.genes);
   para cada i desde 0 hasta lcrom hacer {
      si (alea() < 0.5 ) entonces
         gen = 0;
      eoc
         gen = 1;
      indiv.genes[i] = gen;
   }
   indiv.adaptación = adaptacion(indiv, lcrom);
   devolver indiv;
}
```

2.9.3 Adaptación de los Individuos

En el caso de AG simple, la adaptación de un individuo coincide con el valor de la función objetivo a maximizar. Por lo tanto, el cálculo de la adaptación consiste en decodificar el genotipo del individuo para identificar al punto que representa, y calcular el valor de la función objetivo en ese punto.

Para decodificar un individuo, es decir, buscar el valor real que corresponde a una cadena binaria *c* de longitud *l*, podemos suponer que cada uno de los 2^l números enteros que puede representar la cadena binaria representan un punto del intervalo real $[x_{min}, x_{max}]$ en el que buscamos el óptimo de la función. Podemos considerar estos puntos divisiones de dicho intervalo, que lo particionan en porciones, cada una de las cuales se representa por el valor entero que le corresponde.

Tenemos entonces que el valor real *x* que le corresponde a la división representada por la cadena *c* se puede calcular como:

$$x = x_{min} + posición(c) * tamaño_div$$

La posición de la división asociada a *c* es el valor entero que le corresponde a la cadena binaria considerada y que podemos calcular con una función *bin_ent*, que convierte una cadena binaria en un entero. El tamaño de las divisiones es el mismo para todas ellas y es el que corresponde a dividir en intervalo considerado entre el número de divisiones que podemos hacer, que es el número de enteros de que disponemos, 2^l menos 1.

$$tamaño_div = \frac{x_{max} - x_{min}}{2^l - 1}$$

Tenemos por tanto, que

$$x = x_{min} + bin_ent(c) * \frac{x_{max} - x_{min}}{2^l - 1}$$

Por ejemplo, si tenemos una cadena binaria de longitud 5 de la siguiente forma:

01011

le corresponde el valor entero 11. Supongamos que trabajamos en el intervalo [1,2]. Entonces la cadena es

$$x = 1 + 11\frac{2-1}{2^5 - 1} = 1.354838$$

```
funcion adaptacion(TIndividuo individuo, entero lcrom)
{
  real x; // fenotipo
  real f; // valor de la función a optimizar
  x = decod(individuo.genes, lcrom);
  f = f(x);
  devolver f;
}
```

La decodificación de un individuo se realiza calculando el valor decimal correspondiente al genotipo del individuo, y dividiendo por el número de puntos en el que se ha discretizado el intervalo o espacio de búsqueda considerado.

```
funcion decod(TGenes genes, entero lcrom)
{
  real x;
  x = bin_ent(genes, lcrom) / (pow(2,lcrom) - 1);
  devolver x;     }
```

```
funcion bin_ent(TGenes genes, entero lcrom)
{
   entero i, d=0, pot=1;
   para cada i desde 0 hasta lcrom hacer{
      d = d + pot*genes[lcrom - i - 1];
      pot = pot * 2;
   }
  devolver d;
}
```

2.9.4 Evaluación de la Población

Después de cada generación se revisan los contadores de adaptación relativa y puntuación acumulada de los individuos de la población. Así mismo, se calcula la adaptación global de la población y la posición del mejor individuo.

```
funcion evaluacion(var TPoblacion pob, entero tam_pob,
                   var entero pos_mejor, var real sumadap)
{
  real punt_acu = 0;//puntación acumulada de los individuos
  real adap_mejor = 0; // mejor adaptación
  entero i;
  sumadap = 0; // suma de la adaptación
  para cada i desde 0 hasta tam_pob hacer {
     sumadap = sumadap + pob[i].adaptacion;
     si (pob[i].adaptacion > adap_mejor){
        pos_mejor = i;
        adap_mejor = pob[i].adaptacion;
     }
  }

  para cada i desde 0 hasta tam_pob hacer {
     pob[i].puntuacion = pob[i].adaptacion / sumadap;
     pob[i].punt_acu = pob[i].puntuacion + punt_acu;
     punt_acu = punt_acu + pob[i].puntuacion;
  }
}
```

2.9.5 Selección de Supervivientes

La función de selección del algoritmo genético simple selecciona un número de supervivientes igual al tamaño de la población por el método de la *ruleta*. La función modifica la población que pasa a estar formada únicamente por ejemplares de los individuos supervivientes.

```
funcion seleccion(var TPoblacion pob, entero tam_pob)
{
  real sel_super[tam_pob]; // seleccionados para sobrevivir
  real prob; // probabilidad de selección
  entero pos_super; // posición del superviviente
  TPoblacion pob_aux; // población auxiliar
  entero i;
        para cada i desde 0 hasta tam_pob hacer {
     prob = alea();
     pos_super = 0;
     mientras ((prob> pob[pos_super].punt_acu) y
               (pos_super< tam_pob))
          pos_super++;
     sel_super[i] = pos_super;
  }
  // se genera la población intermedia
  para cada i desde 0 hasta tam_pob hacer {
     pob_aux[i] = pob[sel_super[i]];
  }
  inicializar(pob);
  para cada i desde 0 hasta tam_pob hacer {
     pob[i] = pob_aux[i];
  }
}
```

Probablemente ésta es la forma más simple de implementar la selección, aunque hay muchas otras posibilidades para este operador.

2.9.6 Reproducción, Cruce y Mutación

Consideramos ahora cómo se combinan los operadores de cruce y mutación para producir una nueva generación. La función *reproducción* presenta la secuencia a seguir. En un algoritmo genético simple, la implementación de la reproducción consiste en la selección de los individuos a reproducirse entre los de la población resultante, y finalmente en la aplicación del operador de cruce a cada una de las parejas. Finalmente se aplica la mutación bit a bit mediante la función booleana *mutación*.

```
funcion reproduccion(var TPoblacion pob, entero tam_pob,
                  entero lcrom, real prob_cruce)
{
  real sel_cruce[tam_pob];//seleccionados para reproducirse
  entero num_sel_cruce=0; // contador de seleccionados
  real prob;
  entero punto_cruce;
  TIndividuo hijo1, hijo2;
  entero i;
  // se eligen los individuos a cruzar
  para cada i desde 0 hasta tam_pob {
```

```
      //se generan tam_pob números aleatorios ai en [0 1]
       prob = alea();
      //se eligen los individuos de las posiciones i
      //con ai <prob_cruce
       si (prob < prob_cruce){
         sel_cruce[num_sel_cruce] = i;
         num_sel_cruce++;
       }
    }
    // el número de seleccionados se hace par
    si ((num_sel_cruce mod 2) == 1)
       num_sel_cruce--;
 // se cruzan los individuos elegidos en un punto al azar
    punto_cruce = alea_ent(0,lcrom);
    para cada i desde 0 hasta num_sel_cruce avanzando 2{
       cruce(pob[sel_cruce[i]], pob[sel_cruce[i+1]],
             hijo1, hijo2,lcrom, punto_cruce);
       // los nuevos individuos sustituyen a sus progenitores
       pob[sel_cruce[i]] = hijo1;
       pob[sel_cruce[i+1]] = hijo2;
    }
 }
```

El operador de cruce toma dos padres y genera dos cadenas hijas. Recibe la probabilidad de cruce, junto con la longitud de la cadena. El intercambio de cruce se realiza en un par de bucles. La función calcula la adaptación de los nuevos individuos.

```
 funcion cruce(TIndividuo padre1, TIndividuo padre2,
               var TIndividuo hijo1, var TIndividuo hijo2,
               entero lcrom, entero punto_cruce)
 {
    entero i;
    inicializar(hijo1.genes);
    inicializar(hijo2.genes);
    // primera parte del intercambio: 1 a 1 y 2 a 2
    para cada i desde 0 hasta punto_cruce hacer{
       hijo1.genes[i] = padre1.genes[i];
       hijo2.genes[i] = padre2.genes[i];
    }

 // segunda parte: 1 a 2 y 2 a 1
    para cada i desde punto_cruce hasta lcrom; hacer{
       hijo1.genes[i] = padre2.genes[i];
       hijo2.genes[i] = padre1.genes[i];
    }
    // se evalúan
    hijo1.adaptacion = adaptacion(hijo1, lcrom);
    hijo2.adaptacion = adaptacion(hijo2, lcrom);
 }
```

Finalmente consideramos el operador de mutación, que con la probabilidad indicada considera la posible mutación de cada gen del genotipo. La función revisa la adaptación del individuo en caso de que se produzca alguna mutación.

```
funcion mutacion(var TPoblacion pob, entero tam_pob,
                 entero lcrom, real prob_mut)
{
  booleano mutado;
  entero i,j;
  real prob;
     para cada i desde 0 hasta tam_pob hacer{
     mutado = falso;
     para cada j desde 0 hasta lcrom hacer{
       // se genera un número aleatorio en [0 1)
       prob = alea();
       // se mutan aquellos genes con prob < que prob_mut
       si (prob < prob_mut){
          pob[i].genes[j] = not( pob[i].genes[j]);
          mutado = cierto;
        }
     }
     si (mutado)
        pob[i].adaptacion = adaptacion(pob[i], lcrom);
  }
}
```

2.10 EJEMPLO DE APLICACIÓN A LA BÚSQUEDA DEL ÓPTIMO DE UNA FUNCIÓN

Vamos a aplicar nuestro algoritmo genético a la búsqueda del máximo en el intervalo [0,20] de una sencilla función:

$$f(x) = \frac{x}{1+x^2}$$

La sencillez de la función no justifica la necesidad de aplicación de un AG. Sin embargo, precisamente esta sencillez nos permite utilizarla para clarificar el funcionamiento del AG, que se aplicará exactamente de la misma forma a otras funciones más complejas.

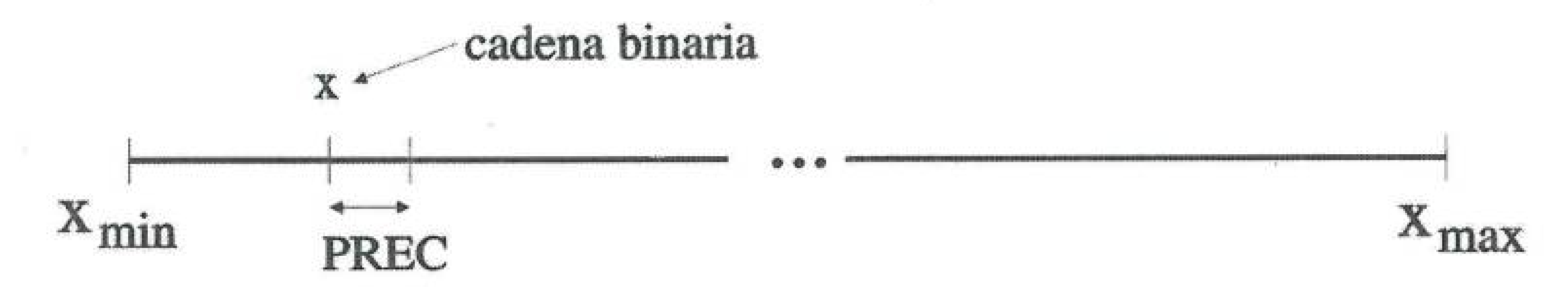
Por los mismos motivos, elegimos también valores de los parámetros que normalmente resultan demasiado pequeños para el buen funcionamiento del AG, pero que nos permiten visualizar el proceso con mayor claridad. Hemos elegido los siguientes valores para los parámetros:

Precisión	0.0001
Tamaño de población	30
Tasa de cruces	40 %
Tasa de mutaciones	1 %
Número de generaciones	10

Suponiendo que trabajamos con una función sobre números reales, necesitamos discretizar dicho intervalo para hacer corresponder posiciones del intervalo real considerado con números enteros representados por nuestra cadena binaria. El parámetro precisión (PREC) indica el número de posiciones decimales en las que nos interesa que el resultado de esta correspondencia sea exacto. Este parámetro condiciona la longitud de la cadena binaria de nuestro algoritmo.

A mayor precisión requerida, más larga debe ser nuestra cadena para poder representar la cantidad de números reales con esa precisión dentro del intervalo de búsqueda especificado. Es habitual utilizar la precisión como parámetro de entrada al AG y a partir de ella calcular la longitud de las cadenas binarias, en lugar de especificar directamente dicha longitud.

Para discretizar el intervalo real contenido entre x_{min} y x_{max}, lo dividimos en pequeñas porciones de anchura menor que PREC, como muestra la siguiente figura:



Los números representados por nuestras cadenas binarias serán las divisiones entre porciones. Con una longitud de cadena binaria l podemos representar a 2^l números enteros, lo que nos permite dividir 2^l-1 porciones de intervalo. Imponiendo que la anchura de las porciones sea menor que PREC tenemos:

$$tamaño\ porción = \frac{x_{max} - x_{min}}{2^l - 1} < PREC$$

De esta fórmula podemos obtener la longitud de cadena binaria que necesitamos para que nuestro algoritmo pueda alcanzar la precisión requerida:

$$l = \left\lceil \log_2 (1 + \frac{x_{max} - x_{min}}{PREC}) \right\rceil$$

donde [x] es el primer valor entero igual o mayor que x.

En nuestro caso, en que x_{min}=0 y x_{max}=2, tenemos

$$l = \left\lceil \log_2 (1 + \frac{20 - 0}{0.0001}) \right\rceil = 18$$

Una vez calculada la longitud de las cadenas binarias de la población podemos generar la población inicial. La siguiente tabla muestra el resultado de esta operación en una ejecución de nuestro algoritmo ejemplo:

Posición	Individuo	x	adaptación
1	000010100010010000	0.709231	0.471874
2	100000010100101011	16.6115	0.059982
3	101110010101101001	11.7698	0.084354
4	001001101111001100	4.05061	0.232694
5	000011011110111101	14.8377	0.0670911
6	001010110011111001	12.4381	0.0798818
7	100000000110111000	2.30477	0.365143
8	110001101101010100	3.34741	0.274262
9	000000110000111110	9.70219	0.101986
10	100010000000101000	1.5638	0.453871
11	000000011011001011	16.5137	0.0603344
12	100111011110100111	17.9634	0.0554968
13	001000111101111111	19.9174	0.0500812
14	010010110110110011	16.0708	0.0619848
15	101111001000110010	5.96171	0.163147
16	001100101101101011	16.7832	0.0593726
17	100101100001010000	0.789264	0.486318
18	010011100010010101	13.2119	0.0752583
19	001000011011000100	2.76399	0.31992
20	110101110001101010	6.7367	0.14524

La tabla presenta para cada individuo su genotipo (la cadena binaria), su fenotipo (el valor real que le corresponde a la cadena binaria) y el valor de adaptación correspondiente al fenotipo.

En este caso la adaptación media de la población es de 0.183415 y el mejor individuo es el de la posición 17, con un valor de adaptación de 0.486318. Podemos observar que en la población inicial los valores de las adaptaciones son muy variados.

Siguiendo con los pasos del algoritmo, la siguiente fase es el proceso de selección de supervivientes por el método de la ruleta. La siguiente tabla muestra en la primera columna los valores generados por este método y la población resultante del proceso.

Selección	Individuo	x	adaptación
11	000000011011001011	16.5137	0.0603344
10	100010000000101000	1.5638	0.453871
19	001000011011000100	2.76399	0.31992
6	001010110011111001	12.4381	0.0798818
15	101111001000110010	5.96171	0.163147
17	100101100001010000	0.789264	0.486318
20	110101110001101010	6.7367	0.14524
3	101110010101101001	11.7698	0.084354
17	100101100001010000	0.789264	0.486318
8	110001101101010100	3.34741	0.274262
20	110101110001101010	6.7367	0.14524
20	110101110001101010	6.7367	0.14524
8	110001101101010100	3.34741	0.274262
12	100111011110100111	17.9634	0.0554968
17	100101100001010000	0.789264	0.486318
17	100101100001010000	0.789264	0.486318
8	110001101101010100	3.34741	0.274262
2	100000010100101011	16.6115	0.059982
17	100101100001010000	0.789264	0.486318
6	001010110011111001	12.4381	0.0798818

Tras el proceso de selección, la adaptación media de la población sube a 0.252348. Podemos observar que los individuos más adaptados, como el 17, tienden a recibir más copias en la nueva población, mientras que los individuos de

baja adaptación, como el 2, tienden a desaparecer. Sin embargo, se trata de un proceso probabilístico, por lo que también los individuos de baja adaptación tienen oportunidades. De hecho podemos ver que el individuo 12, que tiene una adaptación muy baja respecto a la media, ha logrado sobrevivir.

El siguiente paso de la evolución es la reproducción o generación de nuevos individuos mediante el operador de cruce. La siguiente tabla muestra los padres elegidos aleatoriamente para el cruce comprobando la tasa de mutación, y el punto de cruce, también elegido aleatoriamente.

Punto de cruce	padre1	padre2
13	2	3
4	7	12
3	13	15
3	16	17
16	18	19

El estado de la población después de los cruces se muestra en la tabla:

Posición	Individuo	x	adaptación
1	000000011011001011	16.5137	0.0603344
2	100010000000100100	2.81381	0.315537
3	001000011011001000	1.51398	0.459877
4	001010110011111001	12.4381	0.0798818
5	101111001000110010	5.96171	0.163147
6	100101100001010000	0.789264	0.486318
7	110101110001101010	6.7367	0.14524
8	101110010101101001	11.7698	0.084354
9	100101100001010000	0.789264	0.486318
10	110001101101010100	3.34741	0.274262
11	110101110001101010	6.7367	0.14524
12	110101110001101010	6.7367	0.14524
13	110101100001010000	0.789416	0.48634
14	100111011110100111	17.9634	0.0554968
15	100001101101010100	3.34726	0.274272
16	100001101101010100	3.34726	0.274272
17	110101100001010000	0.789416	0.48634
18	100000010100101000	1.61141	0.448032
19	100101100001010011	15.7893	0.0630809
20	001010110011111001	12.4381	0.0798818

En este caso, la adaptación media ha descendido ligeramente, a 0.250673. El mejor individuo en este caso sigue siendo el de la posición 17, y es el producto de uno de los cruces, teniendo un valor de adaptación ligeramente mejor (0.48634) que el de sus progenitores (0.486318 y 0.274262).

Finalmente se aplica el operador de mutación. La siguiente tabla muestra las posiciones de los individuos y posiciones (dentro del individuo) de los genes que han mutado.

Individuo	gen
9	15
10	16
14	8
20	2

El estado de la población después de aplicar el operador de mutación aparece en la siguiente tabla:

Posición	Individuo	x	adaptación
1	000000011011001011	16.5137	0.0603344
2	100010000000100100	2.81381	0.315537
3	001000011011001000	1.51398	0.459877
4	001010110011111001	12.4381	0.0798818
5	101111001000110010	5.96171	0.163147
6	100101100001010000	0.789264	0.486318
7	110101110001101010	6.7367	0.14524
8	101110010101101001	11.7698	0.084354
9	100101100001011000	2.03927	0.395313
10	110001101101010000	0.8474	0.493223
11	110101110001101010	6.7367	0.14524
12	110101110001101010	6.7367	0.14524
13	110101100001010000	0.789416	0.48634
14	100111001110100111	17.9536	0.0555268
15	100001101101010100	3.34726	0.274272
16	100001101101010100	3.34726	0.274272
17	110101100001010000	0.789416	0.48634
18	100000010100101000	1.61141	0.448032
19	100101100001010011	15.7893	0.0630809
20	011010110011111001	12.4382	0.0798808

Tras la mutación, la adaptación media de la población ha pasado a ser 0.257073. El mejor individuo es ahora el 10, que es el producto de una mutación, con una adaptación de 0.493223, un valor que ya se acerca al óptimo de la función que está en 0.5.

La evolución de los valores medio y máximo de la adaptación con las generaciones aparece en la siguiente tabla:

Generación	Adap. Media	Adap. Max.	x
1	0.257073	0.493223	0.8474
2	0.378307	0.498506	0.925525
3	0.395765	0.49323	0.847476
4	0.454126	0.499953	0.986408
5	0.458169	0.499953	0.986408
6	0.467791	0.499953	0.986408
7	0.483441	0.499953	0.986408
8	0.483225	0.499953	0.986408
9	0.489634	0.499997	1.00365
10	0.488957	0.499997	1.00357

Podemos ver como a medida que avanzan las generaciones los valores medio y máximo de la adaptación de la población tienden a mejorar. Sin embargo, la mejora de la adaptación de una generación a otra no está garantizada, a menos que se introduzca elitismo, una técnica que veremos más tarde, para garantizar la supervivencia del mejor, o de algunos de los mejores, de generación en generación. Así, en nuestro ejemplo podemos ver que de la generación 2 a la 3 la adaptación del mejor individuo de la población baja de 0.498506 a 0.49323.

2.11 PROPIEDADES TEÓRICAS DE LOS ALGORITMOS GENÉTICOS

Los algoritmos genéticos, debido a su codificación binaria, presentan un comportamiento diferenciado del resto de los algoritmos evolutivos, que en muchos casos les permite ser más eficientes que un algoritmo evolutivo con otro tipo de representación. Las razones de estas propiedades fueron estudiadas por Holland [HOL75] y la justificación propuesta se basa en considerar que los AGs procesan patrones de coincidencia entre grupos de individuos.

Vamos a comenzar por introducir el concepto de esquema y sus propiedades.

2.11.1 Esquemas

Podemos observar numerosas coincidencias entre las cadenas con las que trabaja un AG. Los patrones de coincidencia entre las cadenas más adaptadas pueden ayudar a guiar la búsqueda. El concepto de **esquema** [HOL75] recoge este hecho.

Definición: un **esquema** es un patrón de coincidencia que se construye sobre el alfabeto {0,1,*}. El símbolo * juega el papel de comodín. Un esquema representa a todos los individuos que coinciden con los ceros y unos del esquema, pudiendo contener cualquier símbolo en las posiciones en las que el esquema tiene el carácter *.

Es decir, al alfabeto binario {0, 1} le añadimos un símbolo especial o comodín, el *. Por ejemplo, el esquema *0000 encaja con las dos cadenas {00000, 10000}, y el esquema 01*000*01 encaja con las cuatro cadenas binarias 010000001, 010000101, 011000001, 011000101. Los esquemas que no contienen el símbolo * sólo representan a una cadena y los esquemas que sólo contienen * representan a todas las cadenas de la longitud del esquema. Puesto que cada comodín * puede tomar dos valores, o 0 ó 1, un esquema con m comodines representa a 2^m esquemas.

El fundamento teórico de los algoritmos genéticos reside en la noción de *esquema*. Para analizar cómo crece y decrece la presencia de los esquemas contenidos en la población de una AG necesitamos formalizar algunas propiedades de los esquemas. Algunos esquemas son más específicos que otros. Por ejemplo, el esquema 011*1** está más definido que el esquema 0******. Además, ciertos esquemas abarcan más parte de la longitud total de la cadena que otros. Por ejemplo, el esquema 1****1* abarca una porción de la cadena mayor que el esquema 1*1****. Para cuantificar estas ideas introducimos dos propiedades de los esquemas: el **orden** de un esquema y la **longitud definida**. Estas propiedades están relacionadas con la representatividad del esquema, y con la probabilidad de que el esquema sobreviva al cruce. Dado un esquema E se definen:

Orden de un esquema, o(E): es el número de posiciones fijadas (es decir, con 0 ó 1) que contiene dicho esquema. Por ejemplo, el orden del esquema 011*1** es 4 ($o(011*1**) = 4$), mientras el orden de 0****** es 1.

Longitud definida de un esquema, *$\delta(E)$***:** es la distancia entre la primera y última posiciones fijadas de la cadena. El esquema 011*1** tiene una longitud definida δ=4, porque la última posición especificada es 5 y la primera es 1, mientras el esquema 0****** tiene longitud de definición δ=0.

Cuanto mayor es el orden del esquema a menos cadenas representa, por lo que el orden es una medida de la representatividad del esquema. La longitud definida mide el grado de agrupamiento o dispersión de la información dentro del esquema. Cuanto menor sea, más agrupada está la información y mayor es la probabilidad de sobrevivir a una operación de cruce.

2.11.2 El Teorema Fundamental

Estas nociones nos permiten formular un resultado fundamental de los AGs, conocido como el Teorema Fundamental de los AGs. Este resultado se refiere a la tendencia que tiene a evolucionar de una determinada forma la presencia de un tipo de esquema en la población. La presencia $m(E,t)$ de un esquema E en la población en un instante t la definimos formalmente como el número de cadenas de la población en el instante t que encajan en el esquema E. La adaptación del esquema E en el instante t, que denotamos por $f(E)$, es la media de las adaptaciones de las cadenas de la población que encajan en el esquema E en el instante t.

Teorema fundamental de los Algoritmos Genéticos: la presencia de un esquema E en la población P de la generación del instante t en un Algoritmo Genético evoluciona estadísticamente de acuerdo con la ecuación:

$$m(E,t+1) \geq m(E,t).\frac{f(E)}{(\bar{f})}\left[1-p_c.\frac{\delta(E)}{l-1}-o(E)p_m\right]$$

donde p_c es la tasa de cruce del AG, y p_m la tasa de mutación. $\delta(E)$ y $o(E)$ son la longitud definida y el orden del esquema E, respectivamente.

Este teorema indica que los esquemas cortos, de orden bajo adaptados por encima de la media, estadísticamente incrementan su presencia en la población de forma exponencial en generaciones sucesivas.

Aunque aquí sólo damos una indicación, la demostración de este resultado es sencilla. Se basa en estudiar el efecto de las operaciones de selección, cruce y mutación sobre los esquemas de la población.

Por ejemplo, el efecto de la selección sobre el número esperado de esquemas en la población se determina de la siguiente forma. Supongamos que en un instante dado de tiempo *t* hay *m* ejemplares de un esquema particular *E* contenido en la población *P(t)*, donde escribiremos *m=m(E,t)* (puede haber distintas cantidades del esquema E en distintos instantes de tiempo). Durante la reproducción, una cadena se copia de acuerdo con su adaptación, es decir, una cadena x_i es seleccionada con probabilidad $p_i = f_i / \sum f_j$.

Tomando una población sin solapamientos de tamaño *n* mediante reemplazamientos a partir de la población *P(t)* esperamos tener *m(E,t+1)* representantes del esquema *E* en la población en el instante *t+1,* como se deduce de la ecuación $m(E,t+1)=m(E,t).n.f(E)/\sum f_j$, donde *f(E)* es la aptitud media de las cadenas representadas por el esquema *E* en el instante *t*.

Si reconocemos que la aptitud media de toda la población puede escribirse como

$$\overline{f} = \sum f_j / n$$

entonces podemos reescribir la ecuación de crecimiento reproductivo del esquema como sigue:

$$m(E,t+1) = m(E,t)\frac{f(E)}{\overline{f}}$$

es decir, la presencia de un esquema particular crece en proporción a la adaptación media del esquema respecto de la aptitud de la población, y por tanto, los esquemas adaptados por encima de la media incrementan su presencia en la población a lo largo de las generaciones. Análogamente se estudian los efectos del cruce y la mutación y se combinan todos ellos.

Dada la importancia de estos esquemas adaptados por encima de la media, de orden bajo y pequeña longitud, se les da el nombre especial de *bloques constructivos*. Se considera que los AGs exploran el espacio de búsqueda a través de la yuxtaposición de este tipo de esquemas, que van aumentando su presencia y recombinándose a lo largo de las generaciones.

2.11.3 Paralelismo Implícito

La razón de la eficiencia de la representación que usan los AGs es que al procesar una población de n individuos, se procesa un número mucho mayor de esquemas. Una estimación al número de esquemas procesados en una población de tamaño n es n^3 [GOL89]. A esta estimación se llega con argumentos estadísticos sobre el número de esquemas de una determinada longitud que hay en una población de tamaño n, y partiendo de los hechos de que:

- En una población de n individuos binarios de tamaño l existen entre 2^l y $n.2^l$ esquemas distintos.

- El número de esquemas de tamaño l_s en un individuo de tamaño l es $2^{ls}(l-l_s+1)$. Para verlo basta considerar el número de esquemas distintos en la primera porción de tamaño l_s del individuo, que es 2^{ls}, y contar el número de desplazamientos de la porción que podemos realizar a lo largo del individuo.

Tomando después un tamaño de población adecuado para que los esquemas de interés no se repitan, se llega a la estimación de n^3 para el número de esquemas procesados.

Podemos ver este resultado como una forma de paralelismo implícito inherente a los AGs que les dota de particular eficiencia.

Sin embargo, hay otros factores a tener en cuenta a la hora de seleccionar el AE que más conviene a nuestro problema. Si la representación binaria es natural al problema y por tanto la codificación y decodificación de los individuos es sencilla, un algoritmo genético será el más adecuado, debido a la propiedad de paralelismo implícito. También es importante que los genes, o bits de los individuos, tengan significado, porque sólo entonces tendrán sentido los bloques constructivos de los que hablábamos. Sin embargo, si la representación binaria no es natural al problema, lo que suele suponer costosas operaciones de codificación y decodificación, y la existencia de bit sin significado en la representación, debemos considerar otras formas más naturales de representación.

Capítulo 3

ALTERNATIVAS A LOS COMPONENTES DE UN ALGORITMO EVOLUTIVO

Los algoritmos evolutivos son una técnica robusta capaz de proporcionar soluciones de calidad con una amplia variación de componentes y parámetros. Sin embargo, también se producen situaciones en las que el algoritmo ya no es capaz de seguir evolucionando, o de hacerlo hacia la dirección adecuada.

Una de las principales propiedades que necesita cumplir un algoritmo evolutivo para producir resultados de calidad es contar con suficiente diversidad en la población. Esta diversidad se refiere tanto a los genotipos de los individuos como a los valores de adaptación que les corresponden. Cuando falta diversidad en los genotipos, los operadores genéticos no son capaces de crear nuevos individuos que se encuentren en una zona del espacio de búsqueda distinta de la de sus progenitores. Así mismo, si los valores que toma la función de adaptación son muy similares, la selección no funciona adecuadamente. Si el valor de adaptación de los mejores individuos se diferencia muy poco del resto, la naturaleza probabilística del mecanismo de selección hará que la composición de la población sea aleatoria, sin conseguir favorecer a los más adaptados. Por otra parte, diferencias demasiado grandes de los valores de adaptación impiden que el mecanismo de selección funcione adecuadamente. El mecanismo de selección hace que los individuos con un valor de adaptación muy superior a la media consigan una cantidad de copias de sí mismos en la siguiente generación muy superior al resto. Esto hace que los restantes individuos tiendan a desaparecer, llegando nuevamente a una situación de falta de diversidad.

En situaciones de falta de diversidad en la población, un algoritmo evolutivo tenderá a converger prematuramente. La convergencia, que fue introducida por De Jong [JON95], se refiere a una evolución en una situación de uniformidad, bien en la composición de los individuos, bien en los valores de adaptación de la población. Una vez que el algoritmo ha convergido, la exploración del espacio de búsqueda se minimiza o se detiene. Si el algoritmo evolutivo funciona correctamente, debe tender a converger hacia el óptimo de los valores de la función de adaptación. Sin embargo, si la convergencia es prematura, es muy probable que el algoritmo no haya conseguido evolucionar hasta tener individuos en la zona del espacio de búsqueda en la que se encuentra el óptimo.

En los algoritmos evolutivos se utilizan medidas de diversidad que permiten detectar situaciones de falta de diversidad, y así poder prevenirlas. Goldberg y Deb [GOL91] comenzaron el trabajo en esta dirección introduciendo el concepto de tiempo de posesión (*takeover time*) como una medida de presión selectiva. La presión selectiva es el grado en el que se favorece a los individuos más adaptados. Es importante que la intensidad de la presión selectiva se encuentre en un rango apropiado, ya que demasiada presión lleva a la pérdida de diversidad, pero poca presión hace que el algoritmo evolucione muy lentamente. El tiempo de posesión mide el número de generaciones necesarias para que toda la población esté compuesta por copias del mejor individuo de la población inicial cuando el único operador que se aplica es la selección. Bäck [BAC94] extendió los resultados del trabajo de Goldberg y Deb analizando y comparando la presión selectiva de los métodos de selección más utilizados. Otra medida muy usada de presión selectiva [HAN94] es la razón entre las probabilidades de selección del mejor individuo de la población y la de la media de la población:

$$presion_selectiva = \frac{Adaptación_máxima}{Adaptación_media}$$

En este capítulo se presentan mecanismos de modificación de la función objetivo para obtener una función de adaptación que favorezca la existencia de diversidad en la población. También se introducirán otros métodos que pueden mejorar el comportamiento de los algoritmos evolutivos en determinadas circunstancias: el elitismo, que garantiza la supervivencia del mejor o de un pequeño grupo de mejores individuos de generación en generación, condiciones de terminación más refinadas y operadores genéticos alternativos a los que utiliza el algoritmo genético simple.

En el ejemplo del capítulo 2 el único objetivo era optimizar la función dada. Sin embargo, es muy frecuente que se presenten problemas en los que, además de querer optimizar alguna forma de función, se desee hacerlo bajo

determinadas condiciones o restricciones. Por ejemplo, en una función de dos variables x e y, podemos desear encontrar valores para estas variables en las que se encuentre el máximo de la función pero siempre que cumplan que $x>y$. En este capítulo también veremos una serie de técnicas generales para abordar problemas con restricciones.

La mayor parte de los mecanismos que se describen en este capítulo son aplicables no sólo a los algoritmos genéticos sino a todos los algoritmos evolutivos. Sólo los que tienen una relación directa con la representación de los individuos, como los operadores genéticos alternativos que se describen, son específicos de los algoritmos genéticos. Sin embargo, hemos decidido abordar en este momento la descripción de estos mecanismos de mejora, antes de pasar a presentar algoritmos evolutivos que utilizan otra representación, porque en este punto del libro el lector ya está preparado para empezar a realizar los proyectos que se presentan en la segunda parte del libro. Por ello es importante que pueda contar con los mecanismos que permiten mejorar los componentes del algoritmo genético simple, que pueden ser necesarios o en todo caso mejorar los resultados de los proyectos abordados.

3.1 DE LA FUNCIÓN OBJETIVO A LA FUNCIÓN DE ADAPTACIÓN

Como decíamos en la introducción, para que el operador de selección funcione correctamente, los valores de la función de adaptación deben tener una separación adecuada: si son muy próximos, la selección no podrá distinguir a los más adaptados para favorecerlos y si se producen valores extremadamente mejores que el resto, la selección puede hacer que la población se sature con copias de estos superindividuos, convergiendo prematuramente. Además, los valores de la adaptación necesitan ser positivos para poder aplicar muchos de los mecanismos de selección que hemos visto.

En esta sección presentamos técnicas que nos permiten convertir nuestra función objetivo en una función de adaptación positiva o en una función que produce valores para distintos individuos con separaciones adecuadas.

3.1.1 Haciendo Positiva la Función de Adaptación

El mecanismo de muestreo que hemos utilizado para la maximización de funciones asigna probabilidades de selección proporcionales al valor de la función de adaptación. Sin embargo, en una minimización lo que nos interesan son los valores más pequeños de dicha función. Luego no se puede aplicar el mecanismo

de muestreo tal cual. Tampoco podemos limitarnos a cambiar el signo de la función, puesto que el mecanismo de muestreo trabaja con valores positivos que representan probabilidades.

La solución está en realizar una modificación de los valores de la función de adaptación, de forma que se obtengan valores positivos y que cuanto menor sea el valor de la función (más cercano al óptimo) mayor sea el correspondiente valor revisado. Sea *cmax* el valor máximo de los valores de adaptación *g(x)* de la población en una determinada generación. Entonces la adaptación revisada *f(x)* correspondiente a la adaptación *g(x)* la definimos como:

$$f(x) = cmax - g(x)$$

que cumple las condiciones requeridas. También podrían utilizarse valores muy superiores a *cmax,* pero entonces la separación relativa de los valores de adaptación de los individuos será menor y el mecanismo de selección funcionará peor.

En lugar de utilizar *cmax* conviene utilizar valores ligeramente mayores (por ejemplo, un 105%) para prevenir que si toda la población converge a un mismo valor de adaptación, la adaptación revisada se haga nula.

Para introducir esta variación, modificamos el código del algoritmo genético tal como aparece en el siguiente pseudocódigo, introduciendo una llamada a la función *revisar_adaptacion_minimizar*, que realiza la modificación de la adaptación para minimización:

```
funcion Algoritmo_Genético_Minimiza(…
{
   TPoblacion pob;     // población
   TParametros parámetros// tamaño población

   obtener_parametros(parametros);
   pob = poblacion_inicial();
   revisar_adaptacion_minimizar(pob, parámetros, cmax);
   evaluacion(pob, tam_pob, pos_mejor, sumadaptacion);

   // bucle de evolución
   mientras no se alcanza la condición de terminación hacer{
      seleccion(pob, parámetros);
      reproduccion(pob, parámetros);
      revisar_adaptacion_minimizar(pob, parámetros, cmax);
      evaluacion(pob, parámetros, pos_mejor, sumadaptacion);
   }
   devolver pob[pos_mejor]
}
```

Esta función calcula las nuevas adaptaciones de cada generación. Como el valor de *cmax* cambia de generación en generación, es necesario revisar todas las adaptaciones. Sin embargo, no es necesario recalcular el valor de la función objetivo porque este no cambia. El valor del fenotipo *x* almacena el valor de la función objetivo, que sólo se calcula al crear un nuevo individuo, bien en la población inicial, bien como resultado de una operación genética, como el cruce o la mutación.

La función de *revisar_adaptacion_minimizar* aparece en el siguiente pseudocódigo:

```
funcion revisar_adaptacion_minimiza(var TPoblacion pob,
                    TParametros param, var real cmax)
{
   cmax = -∞;
   // un valor por debajo de cualquiera que pueda
   // tomar la función objetivo
   para cada individuo desde 1 hasta param.tam_pob hacer{
     si (pob[i].adaptacion > cmax) entonces
        cmax = pob[i].x;
   }
   cmax = cmax * 1.05; //margen para evitar sumadaptacion = 0
                       // si converge la población
   para cada individuo desde 1 hasta param.tam_pob hacer{
      pob[i].adaptacion = cmax - pob[i].x;
   }
}
```

Esta función asigna a *cmax* el mayor valor de la función objetivo en toda la población. El valor de *cmax* se hace ligeramente mayor para prevenir el problema al que se llega si se produce convergencia y toda la población tiene el mismo valor de la adaptación. En este caso, la adaptación revisada de toda la población será cero, y por tanto la suma de las adaptaciones también. Como *sumadaptacion* se usa para calcular la adaptación media, se producirá un error en el algoritmo por división por cero.

Finalmente se calcula la adaptación revisada de toda la población como la diferencia entre *cmax* y la función objetivo.

3.1.2 Escalado de la Función de Adaptación

El escalado de los valores de la función de adaptación nos permite establecer una separación entre los valores que toma para distintos individuos que sea apropiada para el funcionamiento de la selección. Se trata de una contracción o

dilatación del rango de valores que toma la función objetivo. Existen distintas formas de hacer el escalado. Goldberg [GOL89] y Michalewicz [MIC94] consideran los siguientes tipos de mecanismos de escalado:

- **Escalado lineal**.

Con este método la adaptación de un individuo x_i se obtiene a partir de la función objetivo $g(x_i)$ como

$$f(x_i) = a * g(x_i) + b$$

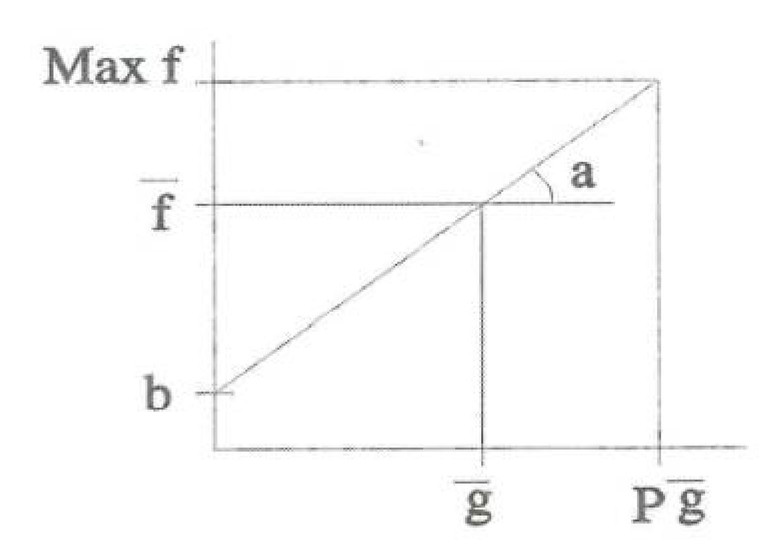
donde a y b se eligen de forma que la adaptación media de la población se corresponda con la media de los valores de la función objetivo:

$$\overline{f} = \overline{g} \quad (1)$$

y que la adaptación del mejor sea el producto por un cierto parámetro P por la adaptación media:

$$f_{max} = P * \overline{g} \quad (2)$$

Es decir, tenemos la situación mostrada en la siguiente figura:



Aplicando la expresión del método de escalado a la ecuación (1) tenemos

$$\overline{f} = \frac{\sum(ag(x_i)+b)}{TamPob} = \frac{a\sum g(x_i)}{TamPob} + \frac{\sum b}{TamPob} = \frac{a\sum g(x_i)}{TamPob} + b$$

Por tanto, tenemos que

$$b = (1-a)\frac{\sum g(x_i)}{TamPob} = (1-a)\overline{g} \qquad (3)$$

Por otra parte, por la ecuación (2) tenemos:

$$f_{max} = ag_{max} + b = P(a\overline{g} + b) \qquad (4)$$

Sustituyendo en esta ecuación b por el valor que hemos obtenido en (3), llegamos a

$$ag_{max} + \overline{g}(1-a) = Pa\overline{g} * P\overline{g}(1-a) \qquad (5)$$

y de aquí podemos obtener el valor de a:

$$a = \frac{(P-1)\overline{g}}{g_{max} - \overline{g}} \qquad (6)$$

De esta forma, una vez fijado el valor del parámetro P, que normalmente toma valores entre 1.2 y 2, tenemos los valores de a y b que nos permiten realizar el escalado.

Para introducirlo en nuestros algoritmos evolutivos, en la función *evaluación* que utiliza el algoritmo después de generar la población inicial, y tras cada ciclo de evolución, llamamos a una función *escalar_adaptación* que calcula los valores de a y b y aplica la transformación de la adaptación para todos los miembros de la población.

```
funcion evaluacion(var TPoblacion pob, entero tam_pob,
                   var entero pos_mejor,var real sumadap)
{
   real punt_acu=0;//puntación acumulada de los individuos
   real adap_mejor = 0; // mejor adaptación
   entero i;
   sumadap = 0; // suma de la adaptación

   escalar_adaptación(pob, tam_pob);

   para cada i desde 0 hasta tam_pob hacer {
      sumadap = sumadap + pob[i].adaptacion;
      si (pob[i].adaptacion > adap_mejor){
```

```
            pos_mejor = i;
            adap_mejor = pob[i].adaptacion;
        }
    }
    para cada i desde 0 hasta tam_pob hacer {
        pob[i].puntuacion = pob[i].adaptacion / sumadap;
        pob[i].punt_acu = pob[i].puntuacion + punt_acu;
        punt_acu = punt_acu + pob[i].puntuacion;
    }
}
```

Los otros métodos de escalado se utilizan de la misma forma.

Dependiendo del rango de valores que tome la función objetivo del problema, este tipo de escalado puede producir valores de adaptación negativos, que tenemos que evitar. En este caso podemos sustituir la segunda condición por otra que especifique que el valor mínimo de la adaptación sea 0:

$$f_{min} = 0 \qquad (7)$$

con lo que se llega al siguiente valor de a

$$a = \frac{\overline{g}}{\overline{g} - g_{min}}$$

- **Escalado Sigma**. Esta técnica, propuesta inicialmente por Forrest [FOR85] hace que el valor de la adaptación de un individuo dependa del valor de la función objetivo, del valor medio de dicha función para toda la población y de la dispersión σ de valores:

$$f(x_i) = \begin{cases} 1 + \dfrac{f(x_i) - \overline{f}}{2\sigma} & si\, \sigma \neq 0 \\ 1.0 & si\, \sigma = 0 \end{cases}$$

siendo σ la desviación estándar:

$$\sigma = \sqrt{\frac{n\sum f^2 - (\sum f)^2}{n^2}}$$

Este mecanismo hace que el comportamiento de la selección varíe a lo largo de la evolución. En las primeras generaciones se tendrán valores altos de la

desviación estándar lo que hará que las variaciones entre las adaptaciones de los individuos sean pequeñas, y las oportunidades de supervivencia se repartan. Según avanza la evolución, la desviación se reduce y los individuos más adaptados tienen más oportunidades.

- **Escalado basado en potencias.** Con este método, propuesto en [MCG85], el valor de la adaptación se obtiene elevando el valor de la función objetivo a alguna potencia k:

$$f(x_i) = (g(x_i))^k$$

Valores de k utilizados en la literatura han sido 1.005 [MIC94].

El método y los parámetros necesarios a utilizar en cada problema es una cuestión de práctica y experiencia.

3.2 ELITISMO

El elitismo consiste en asegurar la supervivencia de los mejores individuos de la población. En su detallado estudio del comportamiento de los algoritmos genéticos en la optimización de una colección de funciones seleccionada a tal fin, De Jong [JON75] descubrió que el elitismo acelera la convergencia de funciones unimodales, es decir, con un único valor óptimo, y por tanto relativamente sencillas. Sin embargo, en funciones multimodales, más complejas, el elitismo puede degradar el comportamiento del algoritmo. Es decir, como De Jong concluyó, "*el elitismo mejora la búsqueda local a expensas de la perspectiva global*".

Por lo tanto, debemos utilizar el elitismo cuidadosamente, teniendo en cuenta las características del problema. En general, el porcentaje de la población que puede formar parte de la élite debe ser pequeño, no mayor de un 1 o un 2% del tamaño de la población. Sin embargo, es una técnica muy útil, que no sólo acelera la convergencia, sino que asegura que si en algún momento de la evolución del algoritmo hemos alcanzado una buena solución, ésta no se perderá en generaciones posteriores.

La forma más sencilla de implementar el elitismo para una élite de E individuos es reservar los E mejores individuos de la generación anterior. Los restantes N-E individuos de la población se seleccionan de la forma usual. Si se utiliza reemplazo inmediato, y los operadores de cruce y mutación se aplican sobre toda la población, incluida la élite, se pueden perder los mejores individuos. Sin

embargo, es importante que los individuos de la élite participen en las operaciones genéticas, ya que son los mejores. Se han propuesto otras alternativas [ZHO05] que tratan de preservar la élite también en las operaciones de cruce y mutación. Aquí presentamos una posible implementación de estas características, ya que la conservación directa de los mejores E individuos durante la selección es muy sencilla y no necesita detallarse más.

En la alternativa que presentamos aquí hemos dado prioridad a conservar la ordenación aleatoria de los individuos de la población. Para implementar el elitismo sin ordenar a los individuos en la población, en la representación de los individuos introducimos un indicativo booleano de pertenecer a la élite. Este indicativo podrá ser consultado por los operados genéticos para respetar de alguna forma a los individuos marcados como élite.

```
tipo TIndividuo = registro{
   TGenes genes; // cadena de bits (genotipo)
      ...
   booleano elite; // indicativo de pertenecer a la élite
}
```

La función de *seleccion* se encarga de poner a verdadero el indicativo booleano de pertenencia a la élite en los individuos de mejor adaptación. Para ello llama a una nueva función *seleccion_elite.*

```
funcion seleccion(var TPoblacion pob, entero tam_pob,
                  entero tam_elite)
{
   real sel_super[tam_pob];//seleccionados para sobrevivir
   real prob; // probabilidad de selección
   entero pos_super; // posición del superviviente
   TPoblacion pob_aux; // población auxiliar
   entero i;

   seleccionar_elite(pob, tam_pob, tam_elite);

   para cada i desde 0 hasta tam_pob hacer {
     si (pob[i].elite = cierto) entonces //ELITISMO
       sel_super[i] = i;
     eoc{
       prob = alea();
       pos_super = 0;
       mientras ((prob > pob[pos_super].punt_acu) y
                (pos_super < tam_pob))
          pos_super = pos_super + 1;
          sel_super[i] = pos_super;
       }
   }
   }
   // se genera la población intermedia
```

```
    para cada i desde 0 hasta tam_pob hacer {
      pob_aux[i] = pob[sel_super[i]];
    }
    inicializar(pob);
    para cada i desde 0 hasta tam_pob hacer {
       pob[i] = pob_aux[i];
    }
}
```

La función *seleccion_elite* se encarga de marcar como élite a los individuos de la población de mejor adaptación. Para ello comienza inicializando a falso el indicativo de élite de toda la población. Después coloca en la lista *sel_elite* las posiciones de los mejores individuos de la población. Para ello empieza colocando en *sel_elite* los primeros *tam_elite* individuos de la población, que serán reemplazados si se encuentran otros mejores. Esta élite inicial se ordena de mayor a menor por las adaptaciones de los individuos. Después se va recorriendo el resto de la población y cada individuo que se encuentra con un valor de adaptación mejor que el de los que ya están en la élite se inserta ordenadamente, desplazando al resto de los individuos que quedan por debajo. Al final de este proceso, *sel_elite* contiene las posiciones de los mejores y se utiliza para poner a verdadero el indicativo de pertenecer a la élite de los correspondientes individuos de la población.

```
funcion seleccion_elite(var TPoblacion pob, entero tam_pob,
                              entero tam_elite)
{
   real sel_elite[tam_pob]; // seleccionados como la élite
   entero i;
   para cada i desde 1 hasta tam_pob hacer
      pob[i].elite = falso;
   para cada i desde 1 hasta tam_elite hacer
      sel_elite[i] = i;
   ordenar_por_adaptación(sel_elite, tam_elite);
   para cada i desde tam_elite + 1 hasta tam_pob hacer {
      j = 1;
      mientras((j <= tam_elite) y
      (pob[sel_elite[j]].adaptacion >= pob[i].adaptacion)){
         j = j + 1;
      }
      si (j <= tam_elite) entonces {
         insertar_ordenadamente(sel_elite, j, i);
      }
   }
   para cada i desde 0 hasta tam_elite hacer {
      pob[sel_elite[i]].elite = cierto;
   }
}
```

Los operados genéticos de cruce y mutación se aplican normalmente, pero si se trata de un AG con estado estacionario en el que los hijos sustituyen a los padres, cuando un hijo va a reemplazar a un padre, se comprueba si el padre pertenece a la élite, y si es así, sólo es sustituido si la adaptación del hijo es mejor que la del padre.

3.3 CRITERIOS DE TERMINACIÓN

En el algoritmo genético simple presentado en el capítulo anterior hemos utilizado como criterio de terminación del algoritmo alcanzar un número máximo de generaciones. Sin embargo existen otras alternativas. Michalewicz [MIC94] considera los siguientes criterios:

- Alcanzar un número máximo de generaciones.
- Alcanzar un número máximo de llamadas al cálculo de la adaptación.
- Llegar a una situación con escasas posibilidades de que se produzcan cambios significativos en la generación siguiente.

Si adoptamos alguna de las dos primeras alternativas necesitaremos alguna estimación preliminar de la complejidad del problema. Podemos conseguirla realizando algunas ejecuciones preliminares del algoritmo con valores tentativos para este parámetro.

El tercer criterio de terminación se basa en el avance que ha conseguido el algoritmo en un cierto número de generaciones. Este número de generaciones también puede ser un parámetro. Las posibilidades de progreso del algoritmo pueden estimarse en función de dos aspectos: la estructura de los individuos, es decir, su genotipo, o su valor de adaptación. Los algoritmos cuya terminación depende del primer aspecto comprueban el número de genes que han convergido a un mismo valor. Se considera que un gen ha convergido si un porcentaje predeterminado de la población tiene el mismo valor para ese gen.

Si el número de genes que han convergido a un mismo valor supera cierto porcentaje, entonces el algoritmo termina. Si se considera el segundo aspecto, es decir, el valor de la adaptación, entonces el algoritmo termina si la mejora de la función de adaptación (la media de la población o el máximo) en un número predefinido de generaciones está por debajo de un cierto valor umbral dado por otro parámetro.

Dependiendo de nuestro conocimiento del problema, podemos utilizar otras condiciones de terminación. Por ejemplo, si se trata de un problema de búsqueda pura, sin optimización, el algoritmo puede terminar al encontrar la primera solución al problema, si es que sólo se necesita una de ellas. También en los problemas de optimización, podemos utilizar criterios basados en las necesidades particulares del caso.

Por ejemplo, si necesitamos encontrar un máximo de una función que esté por encima de un determinado valor, no necesitamos buscar el máximo absoluto, y el algoritmo se puede detener en cuanto el valor de la función objetivo supera el valor buscado.

3.4 VARIANTES DE LOS OPERADORES GENÉTICOS

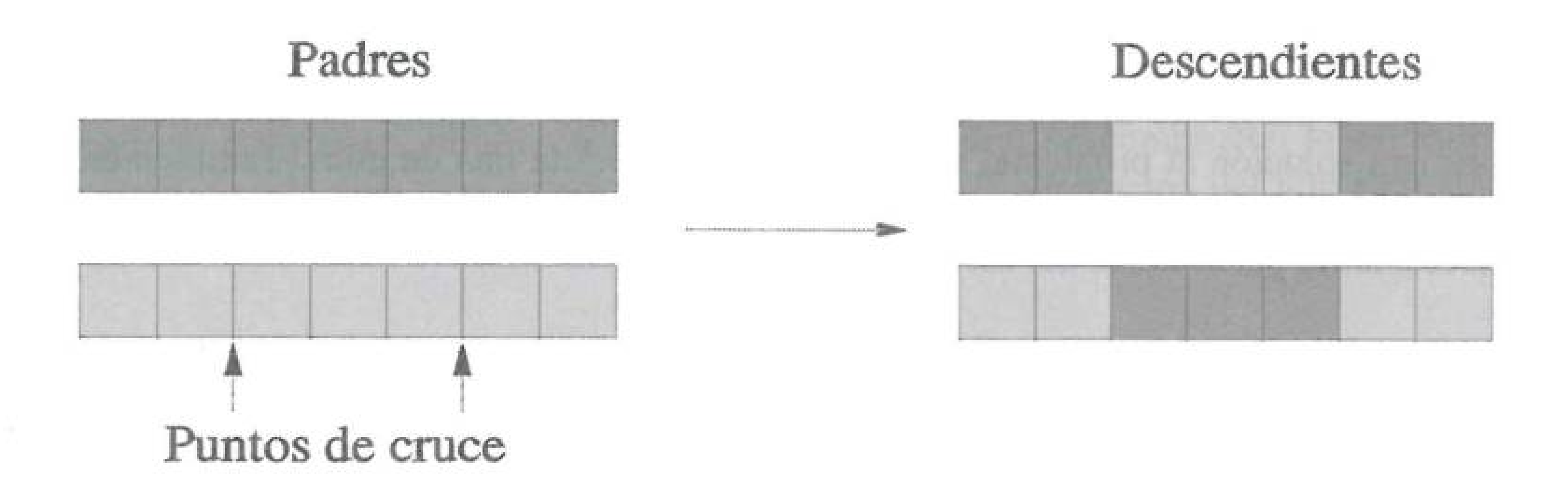
La misión de los operadores genéticos es recombinar la información genética existente en la población y generar nuevo material genético que permita explorar nuevas regiones del espacio de búsqueda. Los operadores de cruce y mutación que utiliza el algoritmo genético simple no son los únicos capaces de crear diversidad.

En esta sección consideramos algunos operadores alternativos al cruce monopunto y a la mutación bit a bit para los algoritmos genéticos. En el capítulo 3 se presentan otros operadores específicos de las representaciones de los individuos utilizadas por cada tipo de algoritmo.

3.4.1 Cruce Multipunto

El cruce monopunto tiene algunos inconvenientes [MIC94] como que no permite cualquier combinación posible del material genético de los padres. Entre los operadores de cruce alternativos propuestos están los de cruce multipunto [ESH89, JON92].

En el cruce de dos puntos se eligen dos puntos de cruce y se intercambia entre los dos padres el segmento definido por dichos puntos de cruce, como muestra la figura:



El cruce de dos puntos se extiende de forma natural al cruce multipunto [ESH89], en el que se tienen múltiples puntos de cruce que definen un conjunto de segmentos que se intercambian entre los padres. Para que este cruce pueda alternar el mismo número de segmentos pertenecientes a cada padre, el número de segmentos debe ser par.

3.4.2 Cruce Segmentado

En el cruce segmentado [ESH89] el número de puntos de cruce puede variar. En este cruce se introduce un nuevo parámetro que es la tasa de segmentado. Este parámetro especifica la probabilidad de que el final de un segmento el individuo esté en un determinado punto de la cadena. Es decir, en cada punto de la cadena determinamos con la probabilidad dada por la tasa de segmentación si el segmento en curso termina en ese punto o se extiende a más posiciones. Para una tasa de segmentado de 0.2 tendremos en promedio 5 segmentos, pero esta cantidad puede variar en cada caso.

3.4.3 Cruce Uniforme

En el cruce uniforme [SYS89, SPE91] se decide con la probabilidad dada por un parámetro de qué padre recibe el primer hijo el valor de cada posición de la cadena. El otro hijo recibe el valor de esa misma posición del otro padre.

3.4.4 Cruce Adaptativo

También se ha propuesto un operador de cruce [SHA87] en el que la distribución de los puntos de cruce se va adaptando a lo largo de la evolución del algoritmo. Para implementarlo se registran los puntos de cruce con los que se ha

realizado el cruce en cada individuo. Los puntos de cruce correspondientes a los mejores individuos tienen más oportunidades de aplicarse.

3.4.5 Tasa de Mutación Variable

También podemos considerar alternativas a la mutación introducida en el algoritmo genético simple. Una variante que se puede introducir en los genéticos es utilizar una tasa de mutación variable que se vaya reduciendo a medida que avanza la evolución [FOG89]. De esta forma la búsqueda es más amplia en las generaciones iniciales y se va reduciendo a medida que avanza la evolución.

Bäck y Schütz [BAC96B] han utilizado un porcentaje de mutación *p(t)*, que se va actualizando de generación en generación de acuerdo con la fórmula:

$$p(t) = \left(2 + \frac{n-2}{T-1}t\right)^{-1}$$

siendo T el número máximo de generaciones, t la generación en curso y n la longitud del cromosoma.

3.4.6 Mutación Adaptativa

En la mutación adaptativa, la tasa de mutación va cambiando con la evolución del algoritmo. La tasa de mutación se incluye en el genotipo, como otra variable que toma valores entre 0 y 1. En este caso, el porcentaje de mutación se agrega como un parámetro más al genotipo, de tal forma que se vuelva una variable más con un valor que oscila entre 0.0 y 1.0.

Bäck y Schütz [BAC96B] han propuesto una transformación de la tasa de mutación de m a m' mediante la siguiente fórmula:

$$m' = \left(1 + \frac{1-m}{m} exp(-\gamma N(0,1)\right)^{-1}$$

donde γ es un parámetro denominado tasa de aprendizaje para el que sugieren un valor de 0.2 y N(0,1) es un número aleatorio normalmente distribuido, con un valor esperado de cero y una desviación estándar de uno.

3.5 TRATAMIENTO DE PROBLEMAS CON RESTRICCIONES

Los problemas de satisfacción de restricciones se formulan sobre un conjunto de variables, cada una de las cuales toma valor en un rango específico, y sobre las que se definen un conjunto de restricciones. El objetivo de la resolución de un problema de este tipo puede ser hallar un valor para cada variable, dentro de su rango específico, de forma que se satisfagan todas las restricciones (problema de satisfactibilidad), o bien buscar una asignación de valores para las variables, que además de satisfacer las restricciones, optimice alguna función objetivo (problema de optimización). Dependiendo del problema los dominios de las variables pueden ser continuos o discretos. Las restricciones pueden involucrar a una, dos o más variables.

Ejemplos clásicos que se utilizan para el estudio de técnicas de este tipo de problemas son los siguientes:

- **Problema de las N reinas**: se trata de colocar N reinas en un tablero de ajedrez NxN, de manera que ninguna de ellas esté amenazada, es decir, no puede haber dos reinas en la misma fila, en la misma columna o en la misma diagonal.

- **Problemas de empaquetado**: se trata de toda una clase de problemas en los que el objetivo es empaquetar un conjunto de objetos en uno o varios recipientes de forma que optimice algún parámetro del empaquetado: minimizar el número de recipientes empleado, maximizar el valor de los objetos empaquetados en un número fijo de recipientes, etc. El problema de la mochila es un caso particular de este tipo de problemas, en el que se quiere maximizar el valor de los objetos empaquetados que se puedan meter en un único recipiente (la mochila). Otro problema de este tipo es el cortado de patrones, para el que se desarrolla un proyecto en la parte II de este libro.

- **Coloreado de grafos o mapas**. El objetivo en este problema es colorear un mapa con un número limitado de colores (o bien minimizando el número de colores empleados) de forma que a todo par de regiones adyacentes (comparte una frontera, no un solo punto) se le asignan colores distintos. El problema se puede formular para un grafo haciendo corresponder cada región con un nodo y las aristas se establecen entre regiones adyacentes.

- **Planificación de horarios y tareas.** El diseño de los horarios admisibles en distintos contextos: colegios, universidades, rutas de autobuses, etc. es un problema muy complejo que se ha tratado en muchos casos con AEs. Lo mismo ocurre con la planificación de tareas que deben desarrollarse en determinados plazos dados por las condiciones del problema o en las que se requieren recursos compartidos cuyo uso en cada momento es necesario planificar.

Aunque en algunos casos se trata de problemas planteados en un marco sencillo, tienen una gran importancia ya que se corresponden con importantes y difíciles problemas reales. Por ejemplo, las soluciones al problema de las N reinas garantizan que las posiciones con reinas pueden accederse desde las ocho direcciones que llevan a sus posiciones vecinas sin conflicto con otras posiciones, y esto tiene aplicación al control de tráfico aéreo, al enrutamiento de mensajes en redes de procesadores, al diseño de rutas de recogida de personas o materiales, etc.

El problema de empaquetado tiene aplicaciones directas a los contenedores de almacenaje y transporte de mercancías, reciclado, etc. El coloreado de grafos tiene también aplicaciones muy importantes como la asignación de frecuencias en telefonía móvil, el diseño de circuitos o la asignación de registros de la máquina por los compiladores de programas. Los problemas de planificación de horarios y tareas tienen aplicaciones evidentes.

Dado la importancia tecnológica de las aplicaciones que se corresponden con este tipo de problemas, se han desarrollado diversas técnicas de trabajo con este tipo de problemas [TSA99, ROS06]. Nosotros consideramos aquí las técnicas basadas en algoritmos evolutivos.

3.5.1 Técnicas básicas

Las técnicas de tratamiento de restricciones en algoritmos evolutivos pueden agruparse en tres tipos básicos:

- **Técnicas de penalización**: son las más generales, ya que pueden aplicarse a cualquier problema con restricciones. Consisten en generar soluciones para el problema ignorando las restricciones y penalizar después en la evaluación a aquellas soluciones que no cumplan las restricciones del problema. A menudo la función de penalización depende del grado de la violación de la restricción, es decir, es alguna función (logaritmo, exponencial, etc.) del grado de la violación. En ocasiones también se hace que la penalización cambie a medida que avanza la evolución, de manera que al comienzo del proceso haya más

permisividad de soluciones que violan las restricciones, y a medida que se acerca al final de la evolución la penalización se incremente.

- **Técnicas de reparación**: son aquellas en las que se busca algún mecanismo para corregir las soluciones que violan las restricciones del problema. Estas técnicas son específicas de cada problema y en general son difíciles de encontrar.
- **Técnicas de codificación**: consisten en buscar una representación especial del problema que garantice que se cumplen las restricciones. Al igual que las técnicas de reparación, son específicas del problema y difíciles de encontrar en general.

3.5.2 Algunos problemas de restricciones tratados con AEs

Michalewicz [MIC94] describe detalladamente cómo se aplican estas técnicas al problema de la mochila. Resumimos aquí la idea de cada uno de los métodos de tratamiento de restricciones para este problema, ya que resulta muy ilustrativo de cada tipo de técnica. El problema de la mochila es un caso particular de problema de empaquetado en el que el objetivo es meter en una mochila, que tiene una determinada capacidad, tantos objetos como sea posible de manera que se maximice el valor del contenido de la mochila. Cada objeto tiene asociado un volumen y un valor. Formalmente, los datos del problema son una mochila de capacidad C, un conjunto de n objetos para los que el volumen del objeto i es $V[i]$ y un conjunto de beneficios de cada objeto ($B[i]$ es el beneficio del objeto i). El objetivo del problema es encontrar un vector binario $x=x[i],...x[n])$, en el que $x[i]=1$ indica que el objeto i se ha metido en la mochila, tal que se cumplan:

$$\sum_{i=1}^{n} x[i]V[i] \leq C,$$

y que,

$$B(x) = \sum_{i=1}^{n} x[i]B[i]$$

sea máximo. Es decir, se pide que los objetos quepan en la mochila y que el valor del contenido sea máximo.

3.5.2.1 TÉCNICAS DE PENALIZACIÓN

Si no consideramos las restricciones del problema, la representación más sencilla de los individuos para este problema es una cadena binaria de longitud n en la que un 1 significa que el objeto se introduce en la mochila. La función de adaptación de cada cadena es la siguiente:

$$f(x) = \sum_{i=1}^{n} x[i] \cdot B[i] - penal(x)$$

La función *penal(x)* debe valer 0 para todos los individuos que respeten la restricción de capacidad, y debe ser mayor que 0 en otro caso. Michalewicz [MIC94] considera tres posibles funciones de penalización:

$$\begin{aligned} Penal_1(x) &= \log_2(1 + \rho \cdot (\sum_{i=1}^{n} x[i] \cdot V[i] - C)) \\ Penal_2(x) &= \rho \cdot (\sum_{i=1}^{n} x[i] \cdot V[i] - C) \\ Penal_3(x) &= (\rho \cdot (\sum_{i=1}^{n} x[i] \cdot V[i] - C))^2 \end{aligned}$$

que se corresponden a tomar una función logarítmica, lineal y cuadrática con respecto al grado de violación. En los tres casos, ρ representa una especie de valor específico de los objetos y se calcula como:

$$\rho = max_{i=1..n}\left(\frac{B[i]}{V[i]}\right)$$

3.5.2.2 TÉCNICAS DE REPARACIÓN

Se pueden implementar diversos procedimientos de reparación para un mismo problema. Michalewicz [MIC94] propone dos de ellos para el problema de la mochila, para el que se sigue utilizando la misma representación de los individuos que en las técnicas de penalización. Ambos se basan en un mismo mecanismo: se comprueba si los objetos incluidos en la mochila exceden su capacidad, y si es así se extraen objetos hasta cumplir la restricción de capacidad. Las dos propuestas se diferencian en la forma de seleccionar los objetos a extraer.

En la primera propuesta (*$Repar_1$*), los objetos a extraer se eligen aleatoriamente. En la segunda (*$Repar_2$*), la selección de los objetos extraídos se realiza con un algoritmo voraz: los objetos de la mochila se ordenan por orden decreciente de valor específico y se extrae el último objeto de la lista.

3.5.2.3 TÉCNICAS DE CODIFICACIÓN

Si recurrimos a las técnicas de codificación necesitamos cambiar la representación de los individuos que hemos estado utilizando hasta ahora para este problema, ya que esta representación no garantiza el cumplimiento de las restricciones. Michalewicz [MIC94] propone una representación en la que cada individuo es un vector de n enteros y el componente de la posición i de este vector toma valores entre 1 y $n-i+1$. Cada componente se interpreta como una referencia a una posición en una lista L de los objetos y la codificación consiste en seleccionar el objeto indicado por cada componente del vector. A medida que se incorporan objetos a la mochila se van extrayendo de la lista L. Los individuos pueden interpretarse como una estrategia para incorporar los objetos a la mochila. El proceso se detiene cuando la incorporación del siguiente objeto incumplirá la restricción de capacidad. Michalewicz propone dos versiones de esta técnica que se diferencian en la forma de ordenar los objetos en la lista L. En la primera versión (*$Codif_1$*) el orden es aleatorio, mientras que en la segunda (*$Codif_2$*) los objetos se ordenan por orden decreciente de valor específico.

3.5.2.4 COMPARATIVA

Para evaluar las distintas técnicas de tratamiento de restricciones, Michalewicz ha utilizado una serie de instancias del problema de la mochila, cuya principal diferencia consiste en el grado de restrictividad del problema: en la versión más restrictiva en la que la mochila tiene una capacidad muy pequeña, la solución óptima contiene muy pocos objetos, mientras que en la versión de capacidad media en la solución óptima están incluidos alrededor de la mitad de los objetos.

Los principales resultados encontrados por Michalewicz son los siguientes:

- Las técnicas de penalización no fueron capaces de encontrar la solución óptima para la versión más restrictiva del problema.

- Para la versión en la que se utiliza una mochila de capacidad media la función de penalización que proporcionó los mejores resultados fue la logarítmica.

- En la versión más restrictiva del problema los métodos de reparación fueron claramente superiores a los restantes.

Aunque el comportamiento de los métodos es muy dependiente del problema y del algoritmo diseñado para cada tipo de técnica, estos resultados nos indican una serie de hechos que concuerdan con nuestra intuición. Es lógico pensar que las técnicas de penalización pueden no llegar a encontrar la solución si el problema es fuertemente restringido y por tanto el espacio de soluciones es muy pequeño. Las técnicas de reparación y de codificación pueden dar muy buenos resultados si el algoritmo correspondiente es natural para el problema y eficiente.

Esto es lo que ocurre en el problema de la mochila con las técnicas de reparación. Por el contrario, las técnicas de codificación propuestas no son tan naturales para el problema, ya que cada posición de la cadena, que ahora es de enteros, deja de tener un significado natural para el problema. La dificultad es que en general es difícil encontrar dichos algoritmos, mientras que las técnicas de penalización son siempre aplicables.

3.5.2.5 EL PROBLEMA DE LAS N REINAS

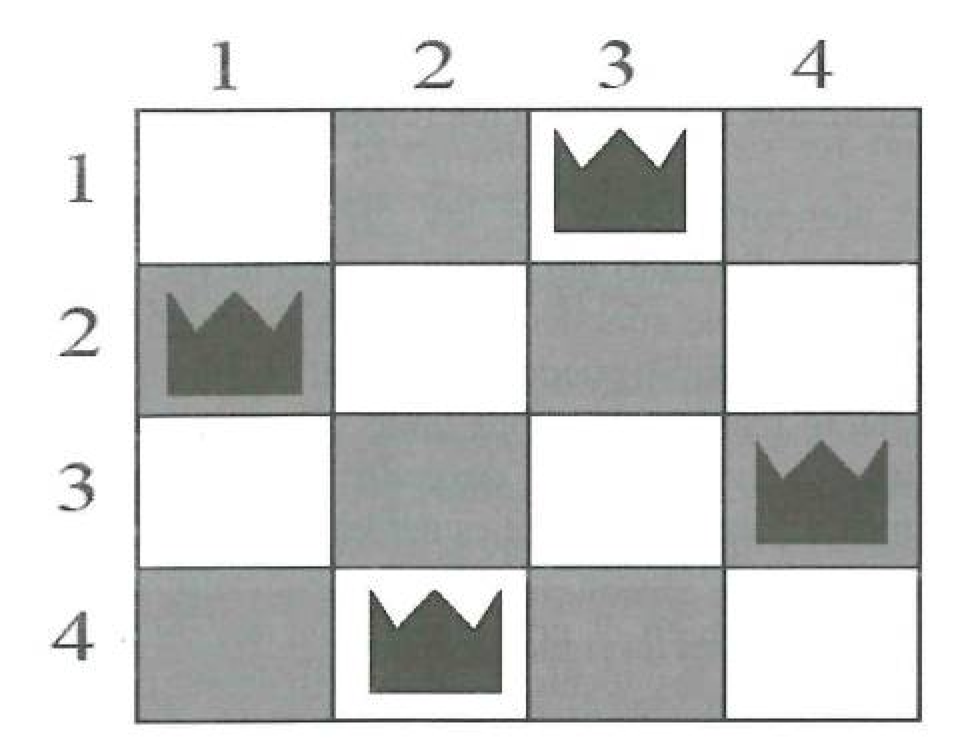
Veamos ahora algunos otros ejemplos de aplicación de algoritmos evolutivos a algunos de los otros problemas de restricciones que hemos mencionado anteriormente.

Resolver el problema de las N reinas significa buscar el valor de N variables $\{R_1, ... R_n\}$, cada una de las cuales toma valores en $\{1..N\}$, y representa la posición de la reina dentro de la columna i. Para que las reinas no ataquen, los valores de estas variables deben cumplir las siguientes restricciones:

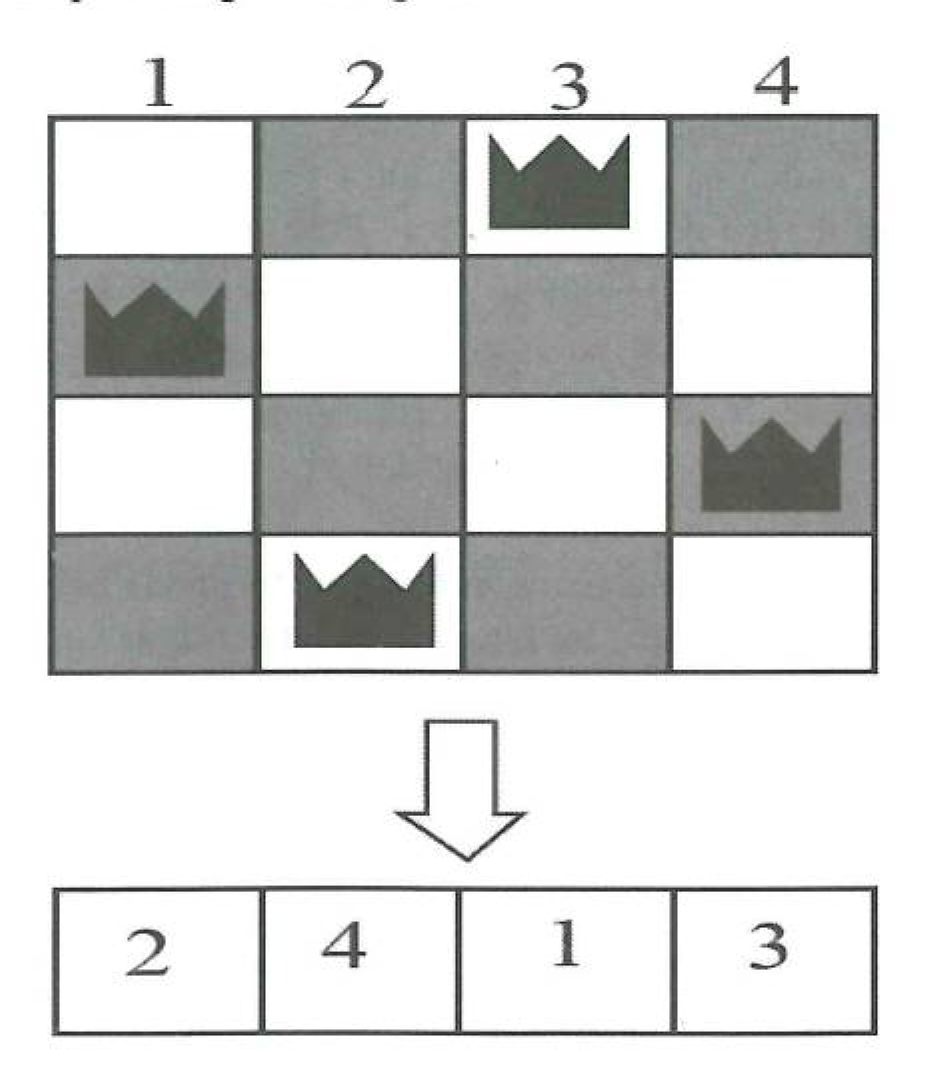
$$R_i \neq R_j$$
$$|R_i - R_j| \neq |i - j|$$

La primera de las restricciones indica que no puede haber dos reinas en la misma columna, y la segunda que no puede haber dos reinas en la misma diagonal, ya sea la diagonal principal o la secundaria.

La siguiente figura muestra una configuración que es solución para el caso de $N = 4$:



Puesto que las restricciones del problema impiden que pueda haber dos reinas colocadas en la misma columna, podemos asignar una reina a cada columna e identificar a cada reina por su fila. Entonces, una sencilla representación de los individuos de un algoritmo genético para este problema consiste en un vector de N enteros en el que el valor de la posición i indica la fila ocupada por la reina que se encuentra en la columna i. Para el ejemplo de la figura anterior tendríamos el individuo mostrado por la siguiente figura:



Con esta representación los operadores de cruce y mutación se pueden definir de forma sencilla. Podemos utilizar un cruce monopunto o de dos puntos. Sin embargo, el simple intercambio de partes de dos soluciones en general da lugar a individuos que no son solución. Por lo que el porcentaje de aplicación de este operador debe ser reducido (Crawford [CRA92]). En cuanto a la mutación, podemos emplear un sencillo intercambio de los valores de dos posiciones. Este operador tenderá a crear nuevas soluciones similares a los padres, quizás con configuraciones simétricas.

El problema de las N-reinas no es por naturaleza un problema de optimización, sino de búsqueda: una configuración de reinas sobre el tablero es solución o no lo es. Pero los algoritmos evolutivos son particularmente efectivos cuando su diseño les permite una aproximación progresiva a la solución. Por ello es conveniente diseñar la función de adaptación de forma que tenga en cuenta tanto como sea posible el grado de acercamiento de un individuo a ser solución.

Crawford [CRA92] propone usar como función de adaptación el número de conflictos a los que da lugar la configuración representada por el individuo. Es decir, para cada reina q_i se comprueba con todas las restantes q_j si se cumple:

$$\| q_i - q_j \| = \| i - j \|$$

y se suma uno en cada ocasión en que no se cumple la condición. Se trata de una minimización y el mecanismo de tratamiento de las restricciones es el de penalización.

Homaifar [HOM92] usa una función de adaptación diferente. En este caso también se comprueba para cada reina el posible ataque de una de las reinas restantes. En este caso en lugar de penalizar los ataques, se premia el no ser atacada. Por cada reina y para cada diagonal se añade una cantidad $1/(2*N)$ a la función de adaptación. En este caso, se trata de una maximización.

Aunque el problema de la N-reinas es un buen ejemplo de problema con restricciones, no es quizás uno de los problemas más apropiados para ser tratados con AEs de forma natural, ya que se trata de una búsqueda y no de una optimización. Sin embargo, podemos ver en las funciones de adaptación que se han propuesto para su resolución cómo podemos transformar la búsqueda en una optimización en la que se va reduciendo el número de violaciones de las restricciones.

3.5.2.6 EMPAQUETADO EN CONTENEDORES

Un posible planteamiento del problema del empaquetado en contenedores, o tridimensional, es el siguiente. Se tiene un conjunto de N objetos, cada uno de los cuales tiene un tamaño, y un conjunto de N contenedores con una determinada capacidad C. El objetivo es encontrar la mejor asignación de objetos a contenedores de forma que los objetos asignados a cada contenedor no excedan su capacidad y de que el número de contenedores utilizado sea mínimo.

La función de adaptación utilizada por Falkenauer [FAL96] es:

$$f(x) = \frac{\sum_{i=1}^{N} (relleno_i / C)^k}{N}$$

donde N es el número de contenedores usados, $relleno_i$ es la suma de los tamaños de los objetos del contenedor i, C es la capacidad de los contenedores y k es una constante mayor que 1. Se trata de una función a maximizar y podemos interpretarla como una medida de la capacidad de explotación de la capacidad de los contenedores.

Las restricciones del problema se tienen en cuenta en la codificación, de forma que no hay individuos que violen las restricciones. Antes de describir la codificación propuesta por Falkenauer, veamos algunas consideraciones sobre los problemas que conllevarán otras codificaciones más clásicas.

Una forma sencilla de representar las soluciones potenciales del problema en un AE es utilizar un vector en el que cada posición corresponde a uno de los objetos que deben ser empaquetados y a cada una de esas posiciones se les asigna un valor que indica a qué contenedor ha sido asignado. Por ejemplo, el siguiente individuo:

1	2	3	4	5	6	7
A	B	C	D	A	E	D

indica que el primer objeto de la colección ha sido asignado al contenedor A, el segundo al B, etc. Sin embargo, esta representación presenta un problema cuando se le aplican los operadores clásicos de cruce y mutación. En el individuo del ejemplo podemos ver que los objetos primero y quinto han sido asignados al mismo contenedor, así como el cuarto y el séptimo. Por definición del problema, la función de adaptación mejora con el tamaño de los grupos, y por tanto el

agrupamiento de dos objetos en un mismo contenedor deberá tender a transmitirse de padres a hijos. Sin embargo, un operador de cruce clásico tiene altas probabilidades de partir el individuo de forma que se separen los objetos primero y quinto del ejemplo, o el cuarto y el séptimo. Podemos pensar en reorganizar el individuo poniendo juntos los objetos que corresponden al mismo contenedor:

1	5	2	3	4	7	6
A	A	B	C	D	D	E

Sin embargo, cuando el número de objetos asignados a un mismo contenedor crece, la probabilidad de que el cruce los separe sigue siendo alta, como ocurre en el siguiente ejemplo:

ACBBBBB

Esta problemática es general de una clase de problemas conocidos como problemas de agrupamiento. Se trata de problemas de optimización en los que el objetivo es agrupar a los elementos de un conjunto en un pequeño número de grupos o familias. En estos problemas la función a maximizar mejora cuando crece el tamaño de los grupos y empeora cuando crece el número de grupos.

Para solucionar el problema de los operadores genéticos, Falkenauer [FAL96] propone una codificación alternativa: la representación estándar descrita se extiende con una parte de grupo. Por ejemplo, el siguiente individuo:

AAABBB

pasa a

AAABBB:AB

donde la parte de grupo es la que aparece después de los dos puntos. Y el individuo

ADBCEB

pasa a ser:

ADBCEB:BECDA

donde podemos observar que los contenedores que aparecen en la parte de grupo no están ordenados.

En esta representación los operadores genéticos trabajan con la parte de grupo de los individuos, mientras la parte estándar dedicada a los objetos sólo sirve para identificar qué objetos van a cada contenedor. Esto significa que los operadores tienen que manejar individuos de longitud variable. Los operadores genéticos siempre generan individuos que respetan las restricciones. Por ejemplo, el cruce inserta grupos (contenedores) de un padre en el otro, después elimina los individuos repetidos y aplica un proceso de búsqueda local para comprobar que se respeten las restricciones de capacidad, añadiendo más contenedores si es necesario. Se consideran tres tipos distintos de mutación: crear un nuevo grupo, eliminar un grupo existente y mover elementos entre grupos. Todos ellos se aplican de forma que se respeten las restricciones de capacidad.

Iiama y Yakawa [IIM03] usan otra representación en la que los objetos asignados a cada contenedor aparecen agrupados, siendo el número de genes de cada individuo variable. Cada grupo de objetos corresponde a un gen. El genotipo representa la secuencia de conjuntos de elementos para cada uno de los contenedores. Por ejemplo, para un caso con 15 objetos el genotipo g_1:

(1,3,10)(2,9,11)(5,7,13,15)(4,6,14)(8,12)

Intercambios en las posiciones de los genes se decodifican a la misma solución. Otro genotipo g_2 como:

(8,12)(2,9,11)(5,7,13,15)(4,6,14)(1,3,10)

corresponde a la misma solución que g_1. En este AG las soluciones generadas por los operadores de cruce y mutación, a diferencia del de Falkenauer, no dependen de la secuencia de genes.

Sean p_1 y p_2 los padres a partir de los cuales se generan los hijos h_1 y h_2. Consideremos el proceso de generación de h_1. Para generar h_2 se intercambian los papeles de p_1 y p_2. Consideremos los siguientes padres:

p_1	(1,3,10)(2,9,11)(5,7,13,15)(4,6,14)(8,12)
p_2	(3,4,12,15)(6,7,11)(9,10)(1,5,8)(2,13,14)

Se empieza seleccionando aleatoriamente algunos contenedores (genes) de p_1, y copiando estos conjuntos de objetos en h_1.

p_1	1,3,10)<u>(2,9,11)</u>(5,7,13,15)<u>(4,6,14)</u>(8,12)

h_1	2,9,11)(4,6,14)

A continuación se toman de p_2 los conjuntos de elementos que no contienen ninguno de los elementos ya tomados de p_1.

h_1	*(2,9,11)(4,6,14)(1,5,8)*

Después, los conjuntos de elementos del resto de los contenedores de p_2 se copian a un conjunto temporal, excluyendo a los elementos que ya se han tomado de p_1.

T	(7)(10)(13)(3,12,15)

Después se aplica un procedimiento de búsqueda local para colocar la mayor cantidad posibles de objetos de *T* en h_1, metiendo el resto en nuevos contenedores, de forma que se respeten las restricciones de capacidad. Es decir, las restricciones se tratan con una técnica de codificación.

Para aplicar el operador de mutación se seleccionan aleatoriamente dos o tres contenedores del padre, y los objetos de estos contenedores se copian a un conjunto temporal *T* de forma individual, es decir, como conjuntos que contienen un único objeto. El resto de los contenedores se copian directamente al hijo. A continuación se aplica un procedimiento de búsqueda local, como en el cruce, para colocar la mayor cantidad posible de objetos de *T* en los contenedores del hijo. Finalmente se meten en los contenedores que sean necesarios los objetos que queden en *T*. Los autores de este trabajo afirman que esta versión del AG obtiene mejores soluciones.

3.5.2.7 COLOREADO DE GRAFOS

Consideremos un grafo *G*, cuyo conjunto de vértices es *V(G)* y cuyas aristas son *A(G)*. Un grafo es k-coloreable si existe una aplicación $\Phi : V(G) \rightarrow 1,\cdots,k$ tal que $\Phi(u) \neq \Phi(v)$ para todo par de vértices *u* y *v* de *V(G)* adyacentes. Si los valores 1,…,*k* representan colores, entonces Φ asigna distintos colores a los vértices adyacentes del grafo. Se trata de un problema de alta complejidad, que tiene una gran relevancia por sus numerosas e importantes aplicaciones.

Se han propuesto diversos algoritmos genéticos [EIB98, CRO05, GAL99, BAR04] para tratar este problema. Describimos aquí uno de los más sencillos y efectivos propuesto por Galinier y Hao [GAL99]. Aunque originalmente se propuso como una combinación de AG y búsqueda local, un trabajo posterior [GLA03] comprobó que incluso sin la parte específica de búsqueda local, el AG es muy efectivo para el problema.

Para resolver el problema del coloreado con el menor número posible de colores, la técnica que consideramos aquí consiste en aplicar un algoritmo de coloreado con un número de colores *k* fijo, que se va reduciendo mientras se encuentren soluciones. Se empieza con un K_0 suficientemente grande, y mientras se pueda resolver el problema del k-coloreado se vuelve a aplicar el algoritmo para *k-1*. De esta forma el problema del coloreado de grafos se reduce a resolver problemas de k-coloreado de dificultad creciente.

En el tratamiento de este problema hay básicamente dos enfoques en función de si el coloreado se considera una asignación de colores a vértices o una partición de vértices en clases de colores, que es el que se describe aquí. En este enfoque se considera que una configuración es una partición de los vértices en clases de color y se necesita un operador de cruce que transmita dichas clases o subconjuntos de ellas.

Los individuos del algoritmo son posibles particiones de *V*, es decir, de los vértices del grafo, en *k* clases. Estas particiones pueden representar coloreados erróneos del grafo. A cada partición se le asigna una penalización igual al número de aristas que tienen ambos extremos en la misma clase de color. Es decir, las restricciones del problema se tratan con técnicas de penalización, y la función de adaptación se define como:

$$f = \sum_{i=1}^{k} | A_i |$$

donde A_i es el conjunto de aristas que tienen ambos extremos en la clase de color i.

Para crear los individuos de la población inicial se usa un algoritmo voraz que empieza con las clases correspondientes a los k colores vacías:

$$V_1, \cdots, V_k = \varnothing$$

A cada paso se elige un vértice v que tenga un número mínimo de clases permitidas (clases que no contienen ningún vértice adyacente a v). Para seleccionar la clase de color de v entre las permitidas, elegimos aquella V_i con mínimo índice i. En general, este proceso no puede asignar todos los vértices. A los vértices pendientes se les asigna una clase elegida aleatoriamente, lo que hace que los individuos de la población inicial sean diferentes entre sí.

Como operador de cruce se utiliza el denominado *cruce voraz de particiones*, que en cada aplicación genera un único hijo. Para construir las clases $V_1,\ldots,V_k$ del hijo, el algoritmo opera pasa a paso. En el paso l, $(1 \leq l \leq k)$ el algoritmo construye la clase V_l de la siguiente forma. Se toma al padre p_1 o al padre p_2 en función de si l es par o impar. En ese padre se elige la clase que tenga mayor número de vértices para que sea V_l, y se borran esos vértices de los padres p_1 y p_2. Al final de los k pasos pueden quedar algunos vértices sin asignar. Dichos vértices se asignan a una clase elegida aleatoriamente.

En lugar de mutación se usa un algoritmo de búsqueda local, que puede consistir simplemente en ir tomando vértices al azar y asignándoles la clase (color) que produce mejor resultado.

Capítulo 4

OTROS TIPOS DE ALGORITMOS EVOLUTIVOS

En los temas precedentes se han estudiado los algoritmos genéticos como técnicas de búsqueda y optimización y se ha profundizado en el algoritmo genético simple junto con algunas alternativas y mejoras de los componentes básicos del algoritmo evolutivo.

En este tema veremos otros tipos de algoritmos evolutivos que siguen manteniendo el esquema evolutivo básico pero utilizan nuevas formas de representación de los individuos y por consiguiente requieren nuevos operadores genéticos adaptados a dicha representación.

Como se ha visto al estudiar el algoritmo genético simple, la representación de los individuos que hemos visto hasta ahora es la que utiliza código binario. En muchos tipos de problemas la representación estándar en binario que utilizan los algoritmos genéticos es demasiado rígida y limitada y surge la necesidad de plantear otras representaciones más naturales, como por ejemplo estructuras de datos específicas para la representación de la información de un dominio concreto.

En muchos casos elegir una representación diferente implica tratar con individuos que representan soluciones claras y sencillas a un problema, que contienen más información útil, aunque esto puede implicar la pérdida de generalidad y del paralelismo asociado a la representación en binario.

En muchos estudios [GOLB90] se ha comprobado que el comportamiento y rendimiento de los algoritmos evolutivos y genéticos depende fuertemente de la representación utilizada.

Michalewicz [MIC96] analiza las ventajas de la incorporación directa de conocimiento específico en la representación; esta variación en algunos de los puntos básicos del algoritmo genético clásico nos lleva a hablar de **algoritmos evolutivos** o programas de evolución como una serie de técnicas que siguen la estructura básica de los algoritmos genéticos pero con algunas variaciones:

- La representación utilizada es más natural, directa y próxima al dominio del problema a resolver, evitando codificaciones complejas y difíciles de decodificar e interpretar.

- Para la representación se utilizan estructuras de datos que permiten representar las soluciones a un problema: vectores, reales, matrices, listas, conjuntos, árboles, etc.

- Se tienen que definir operadores genéticos específicos de la representación utilizada y que generen individuos válidos y correctos, por ejemplo operadores específicos sobre estructuras de datos de tipo matriz.

Lo visto anteriormente lo podríamos resumir utilizando la traducción directa del título del libro de Michalewicz [MIC96], que se considera una guía de referencia indispensable: Algoritmos Genéticos + Estructuras de datos = Programas de evolución.

Un ejemplo que ilustra muy bien el algoritmo evolutivo o programa de evolución es el clásico problema de investigación operativa para resolver la planificación de horarios, que en realidad es una variante del famoso problema de asignación.

El problema se puede resolver utilizando un esquema evolutivo tradicional pero se modifica la representación y los operadores genéticos clásicos para adaptarlos a una representación de los individuos en formato matricial, siendo más natural y más fácil de interpretar que una codificación binaria compleja.

Una posible formulación del problema es la siguiente: disponemos de un colegio con *m* profesores $\{P_1,\ldots, P_m\}$, una serie de *n* turnos o de horas $\{H_1,\ldots,H_n\}$ y una serie de *k* clases $\{C_1,\ldots,C_k\}$.

Se trata de obtener el horario más adecuado (de acuerdo a unos objetivos especificados) que satisfaga una serie de restricciones más o menos severas:

- En todo momento debe haber un solo profesor en cada clase.
- Un profesor no puede estar en más de una clase a la vez.
- El número de profesores desocupados debe ser lo menor posible.
- Se deben distribuir los turnos a lo largo de toda la semana.
- Se deben dejar las tardes libres a los profesores con dedicación parcial.
- Otras restricciones.

Es un ejemplo destacado de representación matricial, donde mediante una matriz representamos las asignaciones de profesores, horarios y clases. En las filas se representan los profesores, en las columnas se representan las horas y los elementos de la matriz hacen referencia a las clases.

Esta nueva representación en forma de matriz implica modificar los operadores genéticos clásicos: cruce a nivel de matrices y mutación a nivel de matriz.

Antes de analizar los nuevos operadores veamos la representación de un individuo como la matriz que refleja una posible solución u horario, siguiendo las pautas descritas anteriormente:

	H_1	**H_2**	**...**	**H_n**
P_1	C_{11}	C_{12}	...	C_{1n}
P_2	C_{21}	C_{22}	...	C_{2n}
...	...	...	...	...
P_m	C_{m1}	C_{m2}	...	C_{mn}

Clase del profesor m en el intervalo horario 2

La función de adaptación "mide" la calidad de un horario teniendo en cuenta todas las restricciones, que se pueden manejar mediante los operadores genéticos, o mediante técnicas de reparación que gestionan las soluciones inconsistentes [COL82], por ejemplo un horario con duplicidades obtenido al realizar un cruce. Como se vio en el capítulo 5, las técnicas de reparación permiten corregir las soluciones no válidas que violan las restricciones de un problema.

Algunos ejemplos de operadores genéticos para este problema son:

- Cruce: en cada matriz correspondiente a los progenitores a cruzar se calcula para cada profesor (fila) la adaptación asociada a esa solución y se intercambian los ***q*** mejores con el otro progenitor, siendo ***q*** un valor que se puede ir variando.

- Mutación de turnos completos: se seleccionan dos columnas de la matriz y se intercambian.

- Mutación de orden ***p***: para la fila correspondiente a un profesor se seleccionan dos sucesiones contiguas de ***p*** elementos y se intercambian.

- Mutación completa de orden ***p***: mutación de orden ***p*** aplicada a todos los profesores.

En el Proyecto 4 se desarrolla un proyecto para la planificación de los horarios de una escuela y en el Proyecto 7 se aborda la resolución del juego del Sudoku mediante un algoritmo evolutivo. Utilizando también una representación matricial, veremos diferentes operadores genéticos aplicados a matrices que representan tableros con posibles soluciones al juego del Sudoku.

En los siguientes apartados analizaremos con ejemplos dos tipos diferentes de algoritmos evolutivos:

- Algoritmos especializados en optimización combinatoria: representación con permutaciones.

- Algoritmos especializados en optimización numérica: representación con codificación real.

Al final del capítulo se analizará la *Programación genética*, que se puede considerar una variante de los algoritmos evolutivos al utilizar estructuras de datos de tipo árbol para la representación de programas que se hacen evolucionar.

4.1 ALGORITMOS EVOLUTIVOS EN OPTIMIZACIÓN COMBINATORIA

Los algoritmos evolutivos y genéticos son métodos de optimización basados en la naturaleza que pueden utilizarse de forma útil en muchos problemas de optimización [ROT06].

Los problemas de optimización combinatoria son muy variados [CRES05], [JON74] y no existe un prototipo, pero lo que sí tienen en común todos ellos es que el orden de los datos, valores o variables depende directamente del problema o del contexto.

En estos problemas normalmente el espacio de soluciones está formado por ordenaciones de números naturales. La representación más adecuada de los individuos es mediante permutaciones de elementos. Si tenemos que representar K variables, normalmente utilizaremos una lista de K enteros en la que cada valor aparece una única vez.

Como este libro pretende tener un enfoque práctico, vamos a hacer un recorrido por varios ejemplos prácticos basados en problemas reales que son bastante representativos de optimización combinatoria.

Un primer ejemplo es el de la "Colocación de bloques", que consiste en la colocación de unos bloques rectangulares dentro de un contenedor de base rectangular aprovechando al máximo la superficie de dicho contenedor.

Se dispone de un contenedor con una superficie rectangular de anchura A y de altura ilimitada, y una lista de n bloques $Ln=(R_1,R_2,...,R_n)$ con $n>=1$. Cada uno de ellos con una longitud menor o igual que A.

El objetivo es colocar los bloques dentro del contenedor, minimizando la altura total alcanzada por los mismos y cumpliendo ciertas restricciones, como por ejemplo que los rectángulos no se pueden solapar, no pueden sobrepasar los límites de la superficie y no se permiten las rotaciones de 90°.

Tal y como se analizó al principio de este capítulo, una forma más natural y directa que la representación binaria de las soluciones es la representación mediante números enteros. A cada uno de los bloques a colocar se le asigna un número entero y el individuo estará formado por una secuencia de números enteros correspondientes a bloques, en un orden correspondiente a las posiciones en la superficie del contenedor.

Un posible individuo sería [1,3,2,4], reflejando que el bloque 1 es el primero en colocarse, luego el 3, luego el 2 y por último el 4. Con esto conocemos la secuencia de entrada de los bloques en el contenedor, pero necesitamos saber cómo se colocan dentro del contenedor; para ello se utilizan diferentes algoritmos en los que no vamos a profundizar. El Proyecto 6 de la segunda parte de este libro está dedicado a resolver una versión de este problema, aplicado al cortado de patrones.

Otro ejemplo muy sencillo del problema que utiliza representación con permutaciones es el problema del cuadrado mágico. Un cuadrado mágico es una cuadrícula de n x n, en la que tenemos que colocar ciertos números que cumplen la siguiente propiedad: la suma de cualquier fila, de cualquier columna y de cualquier diagonal siempre resulta la misma cantidad, conocida como constante mágica.

En general, si el cuadrado es de **n** x **n**, el cuadrado tendrá n^2 casillas y los números que colocaremos serán del **1** a n^2 y la fórmula para encontrar la constante mágica de un cuadrado mágico de orden **n** es:

$$\frac{n(n^2+1)}{2}$$

En un cuadrado mágico de 3 x 3 debemos colocar todos los números del 1 al 9 de forma que la constante mágica sea 15. En la siguiente figura se muestra una solución, en la que se puede comprobar que se cumplen las propiedades:

8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15

Figura 4.1

El individuo que representa la situación anterior puede estar codificado en un vector de 9 posiciones con valores del 1 al 9 sin repeticiones, como se muestra a continuación:

8	1	6	3	5	7	4	9	2

La representación interna del individuo incluye la variable **genes** que aparece en la definición del individuo que representa un posible cuadrado mágico:

```
tipo TIndividuo = registro{

  genes : vector de entero; //vector de valores enteros
  real adaptación; /función de evaluación
  real puntuacion;//puntuación relativa:adaptación/sumadaptacion
  real punt_acu; //puntuación acumulada
  . . .
}
```

Esta disposición de los valores dentro del vector representa una solución al problema del cuadrado mágico de 3 x 3. La simplicidad de esta representación contrasta con la imposibilidad de utilizar los operadores genéticos clásicos, pues se generarían individuos no válidos (por ejemplo, con valores repetidos).

Otro ejemplo ampliamente estudiado es el problema de las 8 reinas que consiste en colocar 8 reinas en un tablero cuadrangular de dimensiones 8x8, de forma que no se encuentren más de una en la misma línea horizontal, vertical o diagonal; es decir, que las reinas no se coman entre sí. En el capítulo 5 se estudia el problema general con N reinas.

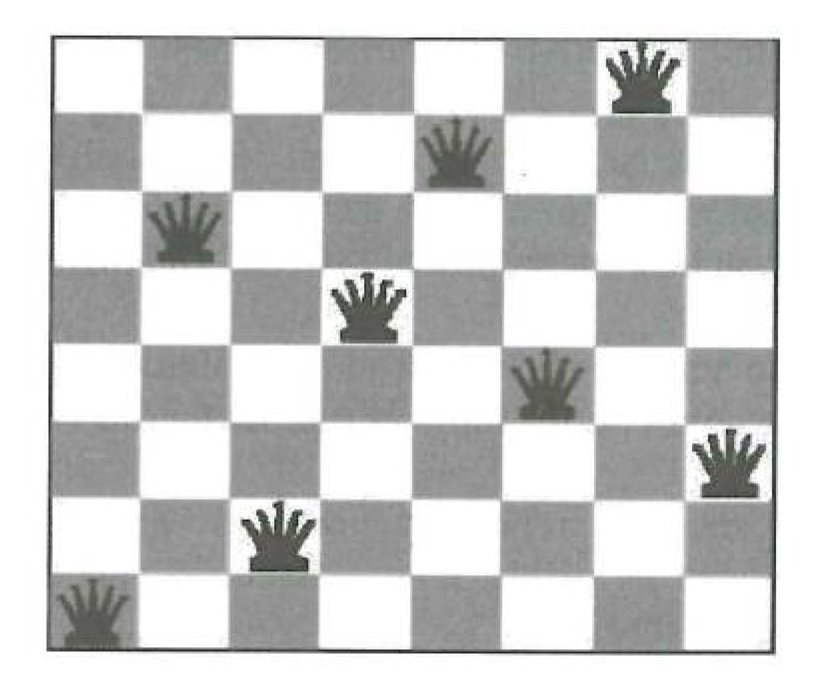


Figura 4.2

Con un enfoque evolutivo, el individuo que representa la situación anterior puede estar codificado en un vector de 8 posiciones con valores del 1 al 8 sin repeticiones, donde cada posición hace referencia a una columna y su valor indica la fila donde está situada la reina:

8	3	7	4	2	5	1	6

En las siguientes secciones estudiaremos operadores genéticos específicos para problemas de optimización combinatoria con permutaciones. Para ello vamos a describir el famoso problema del viajante de comercio que ilustra perfectamente el uso de operadores concretos para representaciones basadas en permutaciones.

4.1.1 El problema del viajante de comercio

Existen muchos otros ejemplos pero entre los más significativos e interesantes a nivel de estudio podemos mencionar el problema del viajante de comercio (TSP en inglés) [GOL90], [APP98].

El problema del viajante de comercio consiste en buscar el recorrido de longitud mínima que tiene que seguir un comerciante para visitar un conjunto de ciudades y volver al punto de partida. La información de la que se dispone es la distancia existente entre ciudades y cumpliendo la restricción de que no se puede pasar dos veces por la misma ciudad.

Estudiaremos a fondo la resolución del problema del viajante de comercio mediante un algoritmo evolutivo por varias razones:

- Es un problema muy explotado dentro del campo de la matemática computacional tanto a nivel teórico como problema NP-completo como a nivel práctico por su utilidad [GAR79], [CRO58].

- Se puede ampliar su significado a cualquier problema similar basado en el orden, como por ejemplo el trazado de rutas o diseño de circuitos.

- Admite muchas representaciones y muchos operadores diferentes dependientes de dichas representaciones, por lo que es muy útil como ejemplo de estudio.

Muchos son los algoritmos genéticos y las aportaciones para resolver este problema: [GRE87], [BAR00].

El problema se formula del siguiente modo:

Dadas n ciudades etiquetadas de **1** a N y las distancias entre unas y otras $\boldsymbol{d_{ij}}$ ($i,j \in 1..N$), se trata de calcular el recorrido más corto que pasa por todas las ciudades y que comienza y termine en la misma ciudad.

Aunque existen muchas formas de codificar las posibles soluciones, la codificación más directa para los individuos consiste en etiquetar las ciudades con números desde 1 a N, de forma que un camino completo será una permutación de los N enteros correspondientes a las N ciudades. Utilizando esta representación y nuevos operadores evitamos tener que utilizar algoritmos de reparación de los individuos que no cumplen las restricciones del problema.

De este modo, el espacio de búsqueda estará formado por todas las permutaciones de las N ciudades y cada una de las permutaciones representa un recorrido candidato a ser la solución del problema. A continuación se analiza más a fondo la representación de las posibles soluciones.

4.1.1.1 REPRESENTACIÓN DE LOS INDIVIDUOS

La manera más natural y directa de codificar las soluciones del problema del viajante consiste en una lista de números en orden correspondientes a las ciudades de un recorrido.

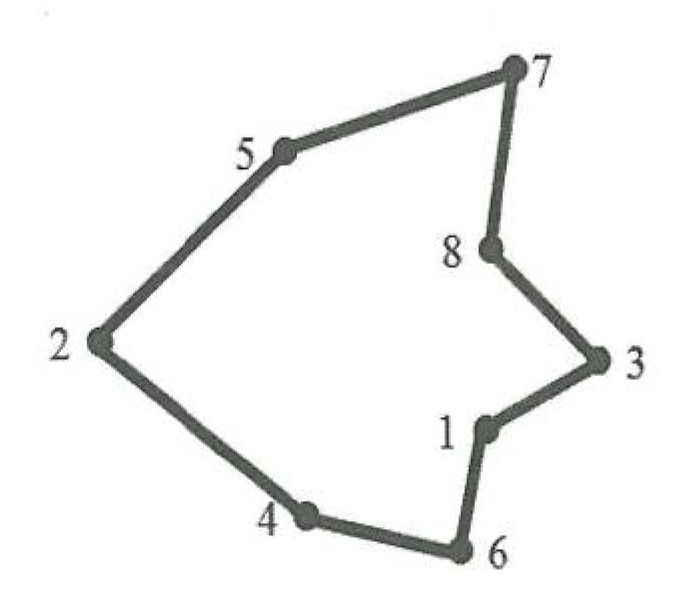
En un problema aplicado a 27 ciudades españolas, la estructura de datos que representa un individuo incluye en este caso información sobre la tabla de distancias entre ciudades, como se muestra en el siguiente pseudocódigo:

```
tipo TIndividuo = registro{

  genes : vector de entero; //identificadores de cada ciudad
  real adaptación; //función de evaluación
  real puntuacion;//puntuación relativa:adaptación/sumadaptacion
  real punt_acu; //puntuación acumulada
  entero longitudCromosoma = 27;
  . . .
  . . .
  Ciudades : vector de cadena_caracteres = {
      "Madrid",
      "Albacete",
      "Alicante",
      "Almería",. . .
  };

 Distancias : matriz de entero = {
            {},
            {251},
            {422, 171},
            {563, 369, 294},
            {115, 366, 537, 663},
            {401, 525, 696, 604, 318},
              . . .
  };
}
```

Para nuestro estudio vamos a simplificar el problema y limitaremos el problema a 8 ciudades etiquetadas con números del 1 al 8, de forma que el siguiente recorrido 7 → 8 → 3 →1 → 6 → 4 → 2 → 5 se representa gráficamente del siguiente modo:



El individuo o recorrido queda simplificado a un vector de 8 números enteros correspondientes a las ciudades:

7	8	3	1	6	4	2	5

Es un recorrido que pasa de la ciudad que está en la última posición (la 5) a la ciudad que está en la primera. En este caso después de visitar la ciudad 5 pasaría a la ciudad de origen, que es la 7.

Esta representación tan sencilla tiene el inconveniente de que no sirve con los operadores genéticos clásicos, pues se generarían individuos no válidos; por ello, haremos un repaso de los operadores genéticos específicos para problemas de optimización combinatoria con permutaciones.

4.1.1.2 OPERADORES DE CRUCE

Cruce por emparejamiento parcial (PMX)

El cruce por emparejamiento parcial [GOL90] consiste en elegir un tramo de la ruta de uno de los padres y cruzar manteniendo el orden y la posición de la mayor cantidad posible de ciudades del otro.

Dados dos padres, este operador de cruce copia una subcadena de uno de ellos directamente en las mismas posiciones del otro. Las posiciones restantes se llenan con los valores que aún no hayan sido utilizados, en el mismo orden que se encontraban en su progenitor.

Los pasos se reflejan en el siguiente pseudocódigo:

```
1. Elegir aleatoriamente dos puntos de corte.

2. Intercambiar las dos subcadenas comprendidas entre
   dichos puntos en los hijos que se generan.

3. Para los valores que faltan en los hijos se copian los
   valores de los padres:

      a) Si un valor no está en la subcadena
         intercambiada, se copia igual.

      b) Si está en la subcadena intercambiada, entonces
         se sustituye por el valor que tenga dicha
         subcadena en el otro padre.
```

Por ejemplo, si tenemos los dos siguientes individuos:

1	3	6	4	2	5	8	7
2	7	8	3	1	5	4	6

Para generar los descendientes, elegimos aleatoriamente las subcadenas comprendidas entre dos puntos de corte que definen los tramos y las intercambiamos:

			3	1	5		
			4	2	5		

En cada uno de los huecos de los hijos colocamos los valores de los padres que no están ya en el hijo:

		6	3	1	5	8	7
	7	8	4	2	5		6

Los valores que faltan se resuelven mediante los valores emparejados en las subcadenas iniciales, obteniendo los dos siguientes individuos:

2	4	6	3	1	5	8	7
1	7	8	4	2	5	3	6

Cruce por orden (OX)

El cruce por orden [DAV85] consiste en copiar en cada uno de los hijos una subcadena de uno de los padres mientras se mantiene el orden relativo de las ciudades que aparecen en el otro padre. El pseudocódigo es el siguiente:

```
1. Elegir aleatoriamente dos puntos de corte.

2. Copiar los valores de las subcadenas comprendidas entre
   dichos puntos en los hijos que se generan.

3. Para los valores que faltan en los hijos se copian los
   valores de los padres comenzando a partir de la zona
   copiada y respetando el orden:

      a) Si un valor no está en la subcadena
         intercambiada, se copia igual.

      b) Si está en la subcadena intercambiada, entonces
         se pasa al siguiente posible.
```

Por ejemplo, si tenemos los dos siguientes individuos:

1	3	6	4	2	5	8	7
2	7	8	3	1	5	4	6

para generar los descendientes elegimos las subcadenas comprendidas entre dos puntos de corte aleatorios:

			4	2	5		
			3	1	5		

Copiamos los valores que nos falten del padre contrario comenzando a partir de la zona copiada y respetando el orden sin repetir elementos:

8	3	1	4	2	5	6	7
6	4	2	3	1	5	8	7

En el apéndice B se incluye código Java correspondiente al cruce por emparejamiento parcial y cruce por orden.

Cruce por ciclos

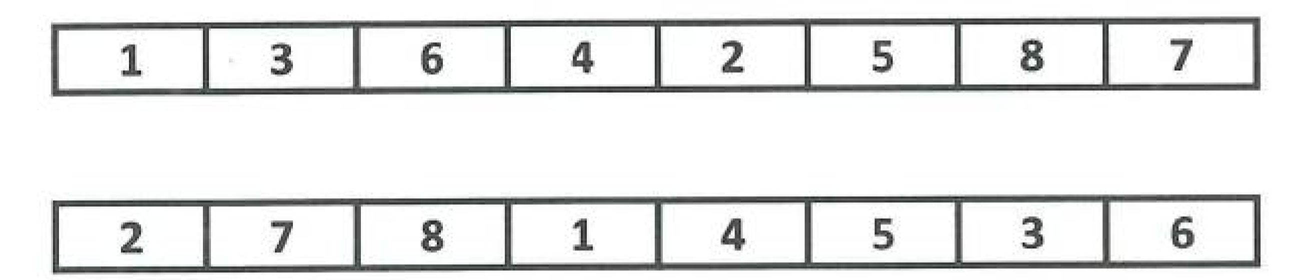
El cruce por ciclos [OLI87] consiste en ir generando los hijos de tal manera que cada ciudad y su posición se va heredando sucesivamente de alguno de los progenitores, de acuerdo con las posiciones que tienen dentro de un ciclo.

El pseudocódigo del algoritmo es el siguiente:

```
1. Encontrar un ciclo que se define mediante las
   posiciones correspondientes de los valores entre Padre
   1 y Padre 2.
```

2. Copiar en Hijo 1 los valores de Padre 1 que sean parte del ciclo.
3. Borrar de Padre 2 los valores que estén en el ciclo.
4. Rellenar el Hijo 1 con los valores restantes de Padre 2 (sustituyendo de izquierda a derecha).
5. Repetir los pasos del 1 al 4, usando ahora el segundo padre.

Por ejemplo, si tenemos los dos siguientes individuos (Padre 1 y Padre 2):



1	3	6	4	2	5	8	7

2	7	8	1	4	5	3	6

empezamos a definir el ciclo por el primer elemento de Padre 1 y alternando con el homólogo en Padre 2 hasta completar dicho ciclo. En este caso el ciclo obtenido es el siguiente: en Padre 1 seleccionamos la ciudad 1 que se corresponde con la ciudad 2 en Padre 2. La ciudad 2 en Padre 1 se corresponde con la ciudad 4 en Padre 2 y la ciudad 4 en Padre 2 se corresponde con la ciudad 1, cerrando un ciclo:

$$1 \rightarrow 2 \rightarrow 4$$

Para generar Hijo 1 copiamos los valores de Padre1 que son parte del ciclo:

1			4	2			

Y para completar Hijo 1 obtenemos las ciudades restantes del otro padre:

1	7	8	4	2	5	3	6

Hijo 2 lo obtenemos del mismo modo:

2	3	6	1	4	5	8	7

Cruce normal con permutaciones

Si en una representación basada en permutaciones aplicamos el cruce normal, obtendremos individuos no válidos. Si queremos utilizar el cruce clásico en un problema basado en permutaciones y asegurar que se obtienen individuos válidos, podemos forzar a una representación ordinal que permita aplicar dicho cruce.

En la representación ordinal, se utiliza una lista variable donde se introducen las ciudades ordenadas según cierto criterio.

Para interpretar y codificar un individuo se van sacando una a una las ciudades recorridas, codificando cada ciudad con la posición que tiene en la lista dinámica, de la que se van eliminando las ciudades consideradas.

Por ejemplo, suponemos que definimos la siguiente lista `{1,2,3,4,5,6,7,8,9}`. Para representar nuestros individuos:

1	3	6	4	2	5	8	7

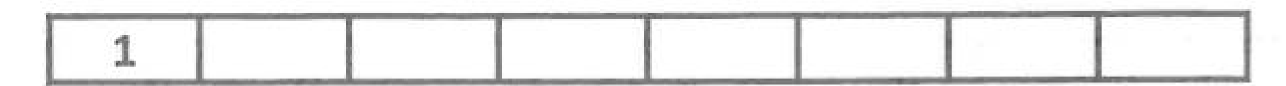
2	7	8	1	4	5	3	6

utilizaremos los valores obtenidos por la posición que ocupa cada ciudad en la lista. Se trata de obtener el lugar de la lista donde está cada una de las ciudades del Padre, colocar ese valor en el hijo y quitar la ciudad de la lista.

Empezamos con la primera ciudad, la 1 y vemos que está en la primera posición de la lista; se saca de la lista y quedaría del siguiente modo:

{ ~~1~~,2,3,4,5,6,7,8,9 } → { 2,3,4,5,6,7,8,9 }

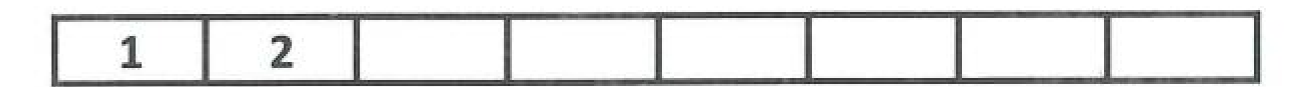
Y se coloca en el hijo el valor de posición donde se encontraba:



La siguiente ciudad del padre original es la 3, que se encuentra en la segunda posición de la lista; al sacarla de la lista, quedaría la siguiente lista:

{ 2,3,4,5,6,7,8,9 } →{ 2,4,5,6,7,8,9 }

Y se coloca en el hijo el valor de posición donde se encontraba (2):



1	2						

La siguiente ciudad del padre original es la 6, que se encuentra en la cuarta posición de la lista; al sacarla de la lista, quedaría la siguiente lista:

{ 2,4,5,6,7,8,9 } →{ 2,4,5,7,8,9 }

Y se coloca en el hijo el valor de posición donde se encontraba (4):

1	2	4					

Repitiendo este proceso hasta el final obtenemos los siguientes hijos:

1	2	4	1	2	2	1	1

2	6	6	2	1	1	2	1

Si ahora aplicamos el cruce clásico en el punto representado por la línea azul, obtenemos los siguientes descendientes:

1	2	4	1	2	2	1	1

2	6	6	2	1	1	2	1

Que al decodificarlos según la lista se corresponden con individuos factibles:

1	3	6	2	5	7	4	8

2	7	8	3	1	4	6	5

4.1.1.3 OPERADORES DE MUTACIÓN

A continuación estudiaremos algunos de los métodos más utilizados como operadores de mutación en representación con permutaciones.

Mutación por inversión

En este tipo de mutación se altera el orden de una subcadena o tramo del individuo. Se aplica con una determinada probabilidad y consiste en seleccionar dos puntos del individuo al azar e invertir los elementos que hay entre dichos puntos.

1	3	6	4	2	5	8	7

1	3	5	2	4	6	8	7

Mutación por intercambio

En este tipo de mutación se seleccionan dos puntos al azar del individuo a mutar y se intercambian los valores de dichas posiciones. Por ejemplo, en el siguiente ejemplo se muestra el intercambio producido entre las posiciones 3 y 7:

1	3	6	4	2	5	8	7

1	3	8	4	2	5	6	7

Mutación por inserción

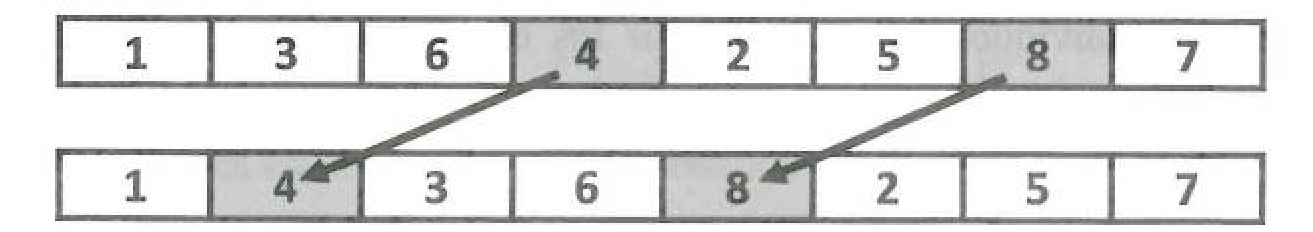
En este tipo de mutación se inserta una o varias ciudades elegidas al azar en unas posiciones también elegidas al azar. Puede haber una o varias inserciones.

Con una sola inserción, por ejemplo, seleccionamos la ciudad 2 y la insertamos en la tercera posición, desplazando todos los elementos hacia la derecha tal y como se muestra en el siguiente ejemplo:

1	3	6	4	2	5	8	7

1	3	2	6	4	5	8	7

Un ejemplo con varias inserciones (también llamada con desplazamiento):



Mutación heurística

En este tipo de mutación se seleccionan **n** ciudades al azar y se generan todas las permutaciones de las ciudades seleccionadas. De todos los individuos que se generan con dichas permutaciones se selecciona el de mejor adaptación. Por ejemplo, si seleccionamos al azar las ciudades 3, 4 y 8:

1	3	6	4	2	5	8	7

las permutaciones con esas ciudades son: 348, 384, 438, 483, 834 y 843, que generan seis individuos, entre los que seleccionaremos el de mejor adaptación:

1	3	6	8	2	5	4	7

1	4	6	3	2	5	8	7

1	4	6	8	2	5	3	7

1	8	6	3	2	5	4	7

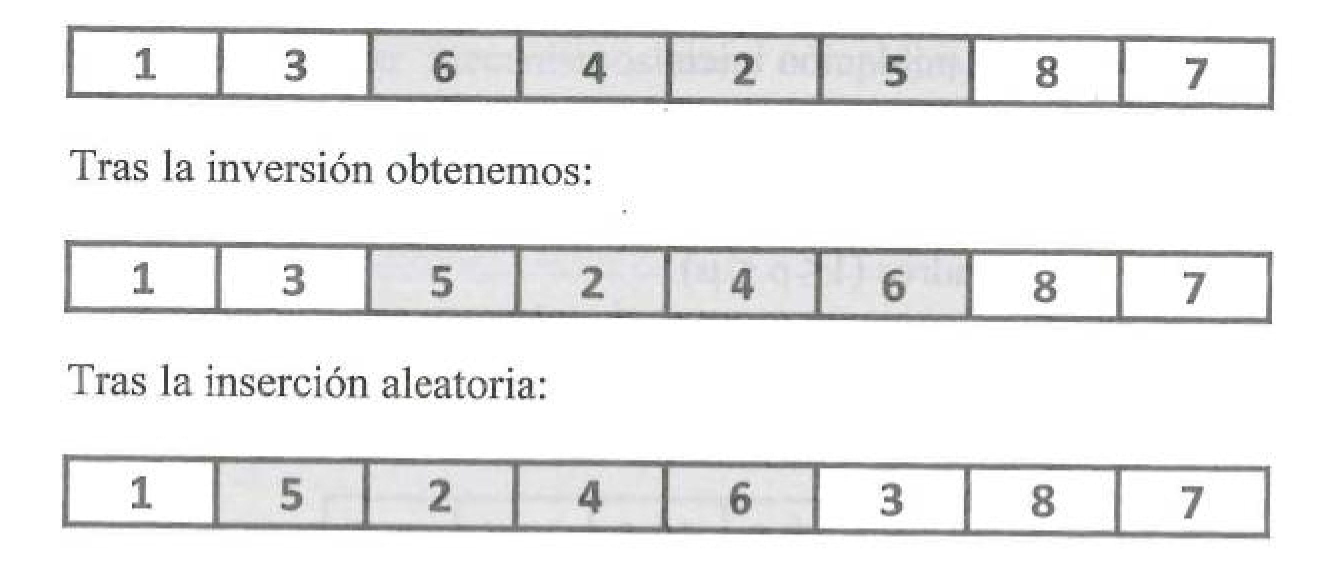
1	8	6	4	2	5	3	7

Mutación combinada

Otro método de mutación consiste en mezclar los operadores de mutación por inserción y por inversión. Este método de mutación consiste en la aplicación de un operador de inversión y a continuación un operador de inserción.

Seleccionamos aleatoriamente un tramo, a continuación se invierten los elementos de dicho tramo según una cierta probabilidad y después el tramo modificado se reinserta en una posición obtenida aleatoriamente, desplazando al resto:

Si este es el individuo a mutar:



4.2 ALGORITMOS EVOLUTIVOS PARA NÚMEROS REALES

Ya hemos visto que en muchos tipos de problemas la representación en binario que utilizan los algoritmos genéticos es demasiado rígida por lo que debemos plantearnos otra codificación.

En problemas de optimización numérica podríamos utilizar el esquema de codificación de valores reales utilizando cromosomas binarios. En este caso podemos fijar los límites inferior y superior para cada variable que queremos representar y su precisión o tolerancia.

Discretizamos el rango de cada variable, fijamos una precisión y tratamos esos valores codificados en binario. Esta representación presenta el problema de cromosomas demasiado grandes en el caso de buscar la codificación de muchas variables con un precisión alta.

En muchos problemas de optimización numérica se ha comprobado la utilidad de utilizar un cromosoma representado como un vector de números reales [HER05], [DEB02], [ESH93]. Esta representación es más cercana al dominio del problema que queremos resolver y permite una mayor precisión numérica.

En los años 60 aparecieron las estrategias evolutivas, desarrolladas por Rechenberg y Schwefel como un método de resolución de problemas de optimización en ingeniería.

En este tipo de estrategias se utilizan los siguientes parámetros:

- μ : tamaño de la población inicial
- λ : tamaño de la población descendente
- ρ : número de padres ($1 \leq \rho \leq \mu$)

Cada individuo de la población se define por el par (x,σ) donde x es un punto en el espacio R definido por $(x_1, x_2, \ldots, x_n)$ y σ es un vector de desviaciones definido como $(\sigma 1, \sigma 2, \ldots, \sigma n)$.

Una primera forma de representación de valores reales consiste en utilizar un valor entero para representar cada uno de los dígitos. Con este esquema podríamos codificar el valor 3.4268 del siguiente modo:

3	4	2	6	8

Con esta representación se podrían utilizar los operadores de cruce clásicos. La mutación podría ser la generación de un dígito aleatorio o la aplicación de una pequeña variación en alguno de los dígitos.

El esquema de codificación que vamos a analizar es la representación de los individuos mediante vectores de números en coma flotante en los que cada componente se corresponde con un gen [DAV91].

3.24	2.71	0.87	2.34	4.55

Con dicha representación, los operadores genéticos varían para adaptarse a esta nueva representación y surgen muchos operadores diferentes. Independientemente del tipo de operador utilizado, la idea fundamental es que normalmente el cruce proporciona dos individuos que "promedian" los valores de sus progenitores y la mutación altera ligeramente el valor del individuo.

A diferencia de los algoritmos genéticos con codificación en binario, con este tipo de representación surge el problema de verificar si los resultados producidos por los operadores genéticos están en sus correspondientes intervalos.

Este problema con los valores obtenidos puede llevar asociado un tratamiento de restricciones que puede consistir en eliminar soluciones no válidos directamente o en aplicar mecanismos más complejos. Esta codificación con valores reales es muy adecuada para resolver problemas con variables en espacios de búsqueda continuos, en los que es importante que los pequeños cambios en las variables impliquen pequeños cambios en la función.

A continuación se muestra la representación y pseudocódigo de un cromosoma o individuo como un vector de números reales, donde cada gen representa una variable del problema:

3.24	2.71	0.87	2.34	4.55

```
tipo TIndividuo = registro{

  genes: vector de real; //vector de valores reales
  real adaptación; //función de evaluación
  real puntuacion; //punt. relativa:adaptación/sumadaptacion
  real punt_acu; //puntuación acumulada
  . . .
}
```

Con este tipo de codificación es importante tener en cuenta que los valores reales que representan los genes tienen que respetar el intervalo de las variables que representan dichos genes.

Por ejemplo, al inicializar el individuo:

```
función inicializaCromosoma()  {
  para i = 0 hasta longitudCromosoma hacer {
    genes[i]=Aleatorio(limiteInf[i],limiteSup[i]);
      . . .
  }
}
```

si este tipo de codificación hace que los operadores genéticos varíen, también es importante que los valores reales que representan los genes respeten el intervalo de

las variables que representan dichos genes y se verifique si los resultados producidos por los operadores genéticos están en sus correspondientes intervalos.

A continuación vamos a estudiar diferentes operadores genéticos aplicables a cromosomas representados mediante vectores de números reales. Son muchos los métodos de cruce que se pueden utilizar:

- Operadores de cruce **discretos**: simple, uniforme, dos puntos.
- Operadores de cruce basados en **agregación**: aritmético, lineal, geométrico.
- Operadores de cruce basados en **entornos**: SBX, BLX.

A continuación se analizan algunos métodos de cruce para representación real. En primer lugar se muestran ejemplos de cruce discreto, donde se selecciona el valor del padre o de la madre con la misma probabilidad.

4.2.1 Operadores de cruce

La notación para representar los padres es la siguiente:

$$P_1 = p_{11}, p_{12}, \ldots, p_{1L}$$

$$P_2 = p_{21}, p_{22}, \ldots, p_{2L}$$

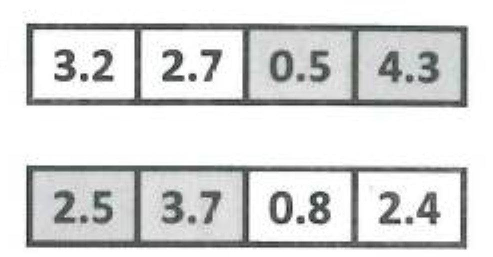
4.2.1.1 CRUCE DISCRETO SIMPLE

Es el cruce de un punto aplicado a vectores de números reales. Por ejemplo si tenemos los siguientes padres y seleccionamos el punto de corte entre el segundo y tercer gen:

3.2	2.7	0.8	2.4

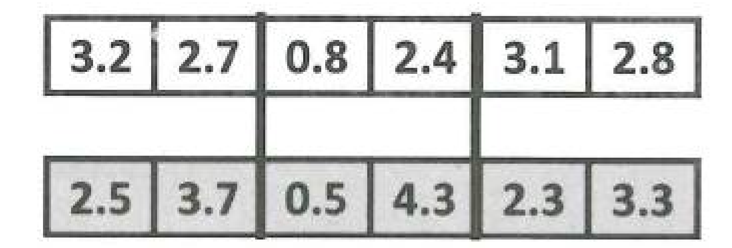
2.5	3.7	0.5	4.3

los hijos obtenidos serían:

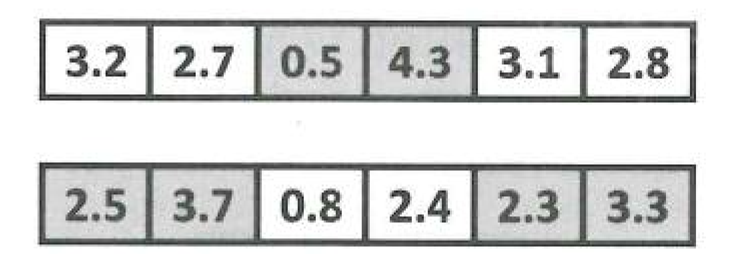


4.2.1.2 CRUCE DISCRETO DE DOS PUNTOS

Es el cruce de dos puntos aplicado a vectores de números reales. Por ejemplo si tenemos los siguientes padres y seleccionamos los dos puntos de corte marcados con líneas:



los hijos obtenidos serían:



4.2.1.3 CRUCE DISCRETO UNIFORME

Se define una probabilidad P_i, que permite sortear cada gen para ver si se intercambian los genes de los padres o se quedan sin cambiar en los hijos. Por ejemplo si P_i es 0,4 y obtenemos los siguientes valores de probabilidad en cada gen:

0.03	0.02	0.6	0.01	0.7	0.8
3.2	2.7	0.8	2.4	3.1	2.8
2.5	3.7	0.5	4.3	2.3	3.3

los hijos obtenidos serían:

2.5	3.7	0.8	4.3	3.1	2.8

3.2	2.7	0.5	2.4	2.3	3.3

4.2.1.4 CRUCE ARITMÉTICO

En este cruce se realiza una combinación lineal entre los cromosomas de los padres. El caso más simple es el cruce de media aritmética, donde el hijo se genera del siguiente modo:

$$h_i = (p_{1i} + p_{2i}) / 2$$

Tiene variantes, por ejemplo se obtiene cada valor del hijo mediante la siguiente fórmula:

$$h_i = \alpha\, p_{1i} + (1-\alpha)\, p_{2i}$$

$$0 <= \alpha <= 1$$

El valor α puede ser constante o variable para cada uno de los valores a calcular. Por ejemplo si tenemos los siguientes padres y el valor de α es 0.6:

3.2	2.7	0.8

2.5	3.7	0.5

(3.2*0.6) + (2.5*0.4) , (2.7*0.6) + (3.7*0.4) , (0.8*0.6) + (0.5*0.4)

(2.5*0.6) + (3.2*0.4) , (3.7*0.6) + (2.7*0.4) , (0.5*0.6) + (0.8*0.4)

los hijos obtenidos serían:

2.92	3.1	0.68

1.78	3.3	0.62

4.2.1.5 CRUCE MEDIA GEOMÉTRICA

Para generar el cromosoma hijo se utiliza la siguiente fórmula:

$$h_i = (p_{1i} + p_{2i})^{1/2}$$

Tiene el inconveniente de que se pierde diversidad ya que tiende a llevar los individuos hacia la mitad del intervalo permitido $[a,b]$.

4.2.1.6 CRUCE SBX

El cruce binario simulado SBX [DEB95] se aplica generando un número aleatorio α entre 0 y 1. Este valor α determina el modo de calcular otro valor β.

Si $\alpha < 0.5$ entonces $\beta = 2\alpha^{(1/(n+1))}$

Si $\alpha >= 0.5$ entonces $\beta = (1/(2(1-\alpha)))^{(1/(n+1))}$

El valor recomendado para n es 1 ó 2. El valor de cada hijo se puede obtener mediante la siguiente fórmula, para cada uno de los hijos:

$$h1_i = 0.5\ (\ (P1_i+P2_i)\ -\ \beta|P2_i-P1_i|)$$

$$h2_i = 0.5\ (\ (P1_i+P2_i)\ +\ \beta|P2_i-P1_i|)$$

4.2.1.7 CRUCE BLX-α

Este operador [ESH93], también llamado de mezcla, permite mantener la diversidad. Se generan dos números aleatorios α y β entre 0 y 1.

Cada valor del hijo se puede obtener mediante la siguiente fórmula:

$$h_i = r_i + \beta(r_i - s_i)$$

donde:

$$r_i = \min(x_i, y_i) - \alpha|x_i - y_i|$$

$$s_i = \max(x_i, y_i) - \alpha|x_i - y_i|$$

El valor α genera valores para el hijo dentro del intervalo de valores de los padres.

4.2.2 Operadores de mutación

4.2.2.1 MUTACIÓN UNIFORME

En el caso de la mutación uniforme sobre valores reales la transformación consiste en, dado un individuo, modificar alguno de los valores cambiándolo por un valor aleatorio entre los posibles del intervalo:

$$P = \{V_1, V_2, \ldots\ldots, V_m\}$$

Obtener el nuevo individuo variando un valor V_r:

$$P = \{V_1, V_2, \ldots, V'_r, \ldots, V_m\}$$

donde V'_r = aleatorio(Inf_i, Sup_i).

Por ejemplo, si tenemos el siguiente individuo:

3.2	2.7	0.8	3.8

tomamos como valor a modificar $V_k = 0.8$:

$Inf_3 = 0.3$

$Sup_3 = 6.1$

Se obtiene $V'_k = 2.3$, obteniendo el nuevo individuo:

3.2	2.7	2.3	3.8

4.2.2.2 MUTACIÓN NO UNIFORME

Normalmente se aplican pequeños cambios, añadiendo a cada variable una pequeña desviación aleatoria de una distribución normal N (0, σ), donde el valor de σ (desviación estándar) es el que controla el grado de cambio aplicado.

4.3 PROGRAMACIÓN GENÉTICA

Desde sus inicios, uno de los principales objetivos dentro del campo de la inteligencia artificial ha sido el conseguir que las computadoras resuelvan

problemas sin programarlas explícitamente para ello. Podríamos hablar de programación automática, método por el cual sólo le indicamos a la máquina lo que hay que hacer pero no le indicamos los pasos para llevar a cabo dicha tarea.

Una forma de llevar a cabo esta "Programación automática" es con el paradigma de la Programación Genética [KOZ92].

En Programación genética se trabaja con poblaciones de programas que se hacen evolucionar hasta conseguir uno suficientemente bueno para resolver un determinado problema. En muchos casos, un programa es la mejor forma de representar la solución a un problema, pues dicho programa puede ser simplemente la secuencia de pasos para conseguir un objetivo.

El proceso evolutivo en programación genética involucra las operaciones genéticas habituales de selección, cruce y mutación y está guiado por una medida de la adaptación de los programas frente al problema a resolver. Más adelante se analizan los operadores genéticos aplicados a esta nueva representación.

Los programas con los que trata la programación genética suelen estar formados por un conjunto de funciones y de símbolos terminales propios del dominio del problema. Para crear estas combinaciones de funciones y símbolos es muy apropiada la sintaxis del lenguaje LISP, que utiliza expresiones fácilmente representables mediante estructuras tipo árbol, donde los nodos representan funciones y las hojas representan símbolos terminales.

Podíamos resumir el algoritmo de programación genética en los siguientes pasos:

```
1. Generar una población inicial de programas aleatorios,
   mediante expresiones LISP o árboles formados por
   funciones y símbolos terminales.

2. Repetir hasta Fin:

     a. Ejecutar cada programa y calificar su adaptación
        para resolver el problema.
     b. Aplicar selección.
     c. Aplicar cruce y/o mutación.

3. Devolver el mejor programa como solución.
```

En la mayor parte de las aplicaciones, la adaptación de un individuo se calcula sobre una colección de instancias particulares del problema considerado.

Debido a que la represenetación más usual en programación genética es un árbol, hablaremos de funciones y terminales, que representan respectivamente los nodos internos y las hojas del árbol.

Los pasos a seguir en un problema de programación genética son los siguientes:

- Identificar el conjunto de terminales
- Identificar el conjunto de funciones y su aridad
- Identificar la función de adaptación
- Identificar los parámetros del algoritmo
- Identificar los criterios de terminación

La identificación de terminales y funciones obviamente depende del problema. Si el problema consiste en mover un elemento por un tablero, las funciones podrían ser sentencias condicionales o sentencias que agrupen otras sentencias o simulen repetición y los terminales podrían ser los propios movimientos sobre el tablero, nombres de variables independientes o constantes.

La función de adaptación normalmente consiste en evaluar la "calidad" de un programa o individuo respecto a la resolución del problema. Por ejemplo, si el problema consiste en recorrer el número máximo de casillas de un tablero, valoraríamos el número de casillas recorridas al ejecutar el programa en cuestión.

A la hora de configurar los parámetros del algoritmo deberíamos establecer el tamaño de la población (número de programas que evolucionan), las probabilidades de aplicación de los operadores genéticos, la profundidad de los árboles o longitud máxima de los programas, etc. La identificación de los criterios de terminación puede implicar el número máximo de generaciones, o la calidad alcanzada por una solución.

Para ilustrar los conceptos de programación genética nos basaremos en un ejemplo práctico [KOZ90]. Queremos calcular la fórmula o programa para calcular el período orbital de un planeta P, dada su distancia media al sol A. La tercera ley de Kepler establece que

$$\mathbf{P^2 = cA^3} \quad \text{siendo } \mathbf{c} \text{ una constante.}$$

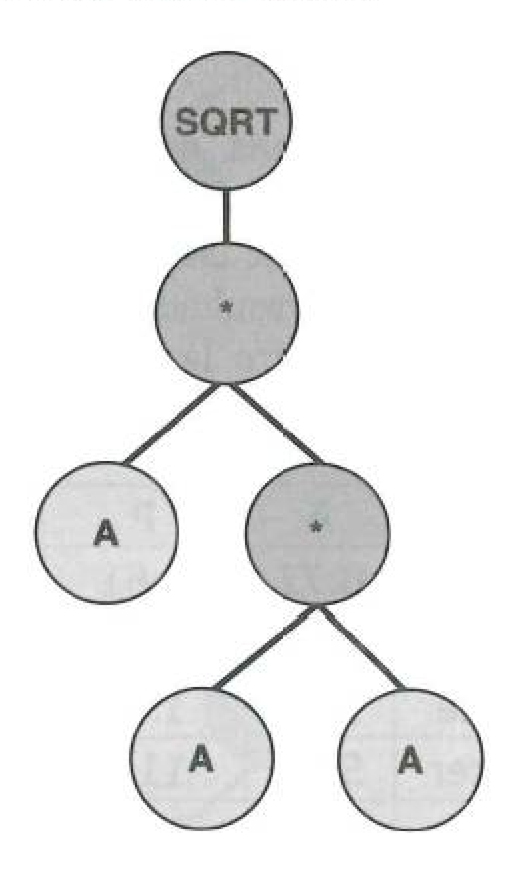
Suponiendo que P se expresa en años terrestres y A se expresa en unidades de la distancia media de la tierra al sol, c toma el valor 1. Por lo tanto:

$$P = \sqrt{A^3}$$

que se representa mediante la siguiente expresión:

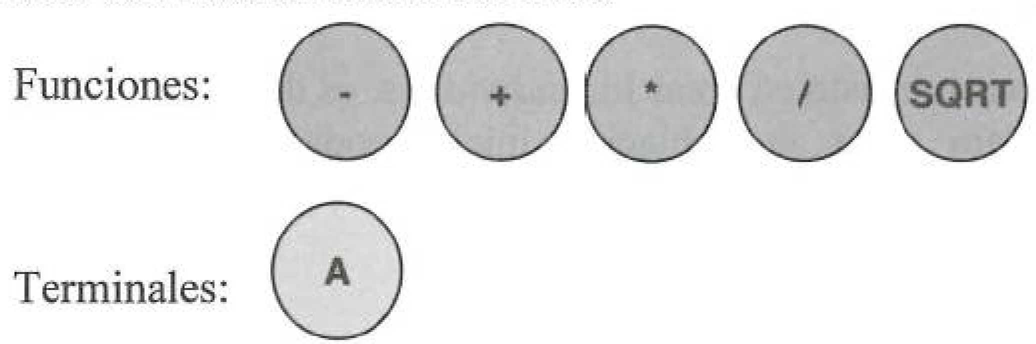
P = sqrt (A*A*A)

Por lo tanto esa es la expresión que queremos obtener o descubrir automáticamente a partir de datos medidos para P y A y que se puede representar fácilmente mediante una estructura de árbol:



En primer lugar debemos elegir el conjunto de posibles funciones y terminales para el programa. No sabemos a priori exactamente las funciones y terminales que se van a necesitar para que el programa tenga éxito, pero podemos considerar un conjunto de posibles candidatas.

En este caso el conjunto de funciones puede ser +, -, *, /, sqrt y como terminales sólo consideramos la variable **A**, al deducir de la especificación que la función será una función aritmética de A.



Se genera un población inicial de programas aleatorios (árboles) sintácticamente correctos, en los que cada operador tiene los hijos que le corresponden, usando el conjunto de funciones y terminales posibles y utilizando alguno de los métodos que se describen más adelante.

A continuación se muestran ejemplos de programas:

```
(+A(*(SQRT A) A))

(*A(-(*A A) (SQRT A)))

(*A(-(* SQRT A A) (+ A A)))

. . .
```

Para analizar la adaptación de cada uno de los programas generados calculamos su adaptación sobre un conjunto de datos de entrada de P y A como medidas experimentales; así, la adaptación de un programa puede ser el número de casos de prueba en los que produce un resultado correcto o con un error muy pequeño. Realmente es medir el error entre las salidas obtenidas y las salidas deseadas.

	A	P
Venus	0.72	0.61
Tierra	1.0	1.0
Marte	1.52	1.84
Júpiter	5.20	11.9
Saturno	9.53	29.4
Urano	19.1	83.5

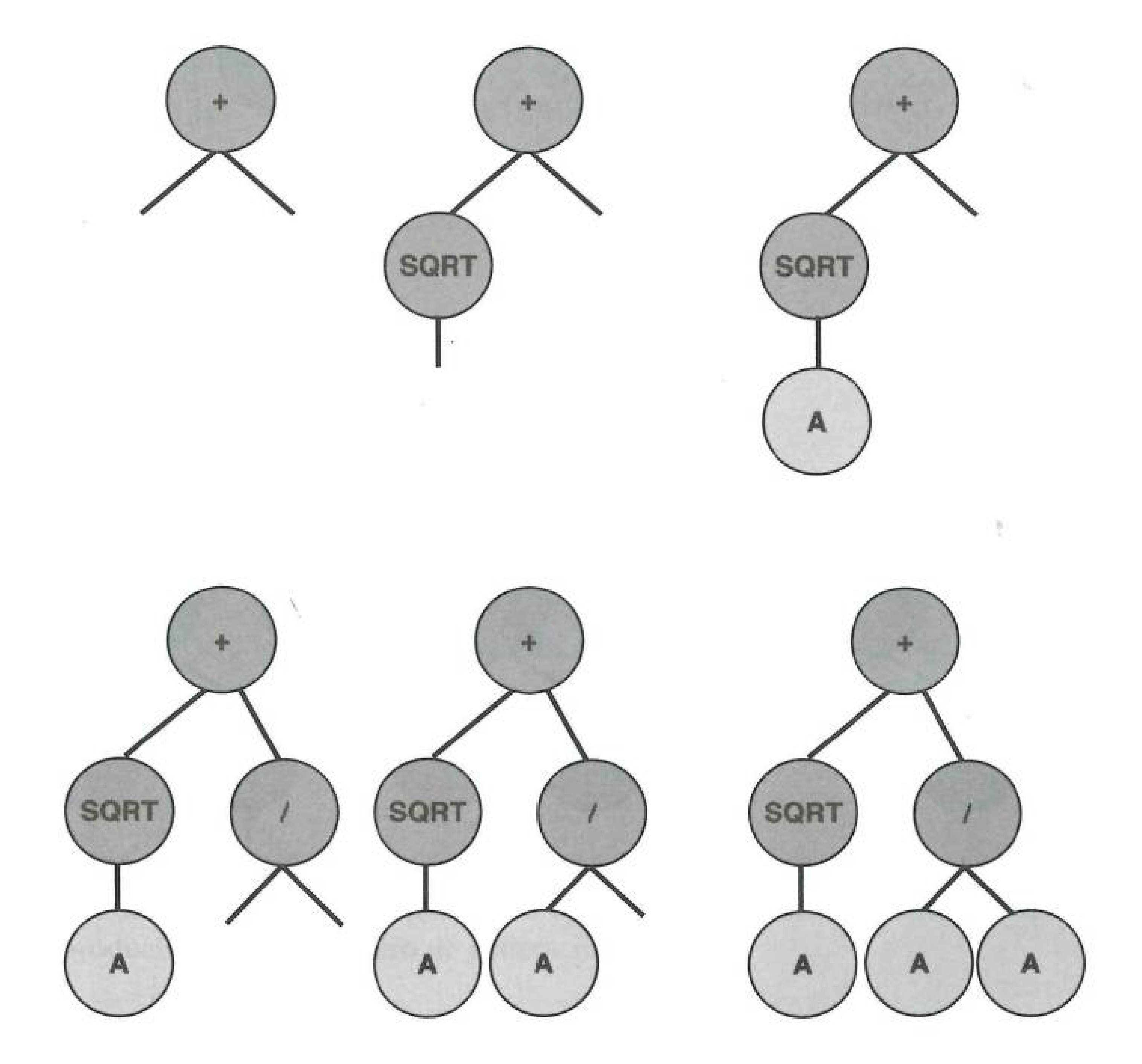
Con una configuración de los parámetros del algoritmo se aplican los operados de selección, cruce y mutación para producir un cierto número de generaciones.

4.3.1 Creación de los individuos

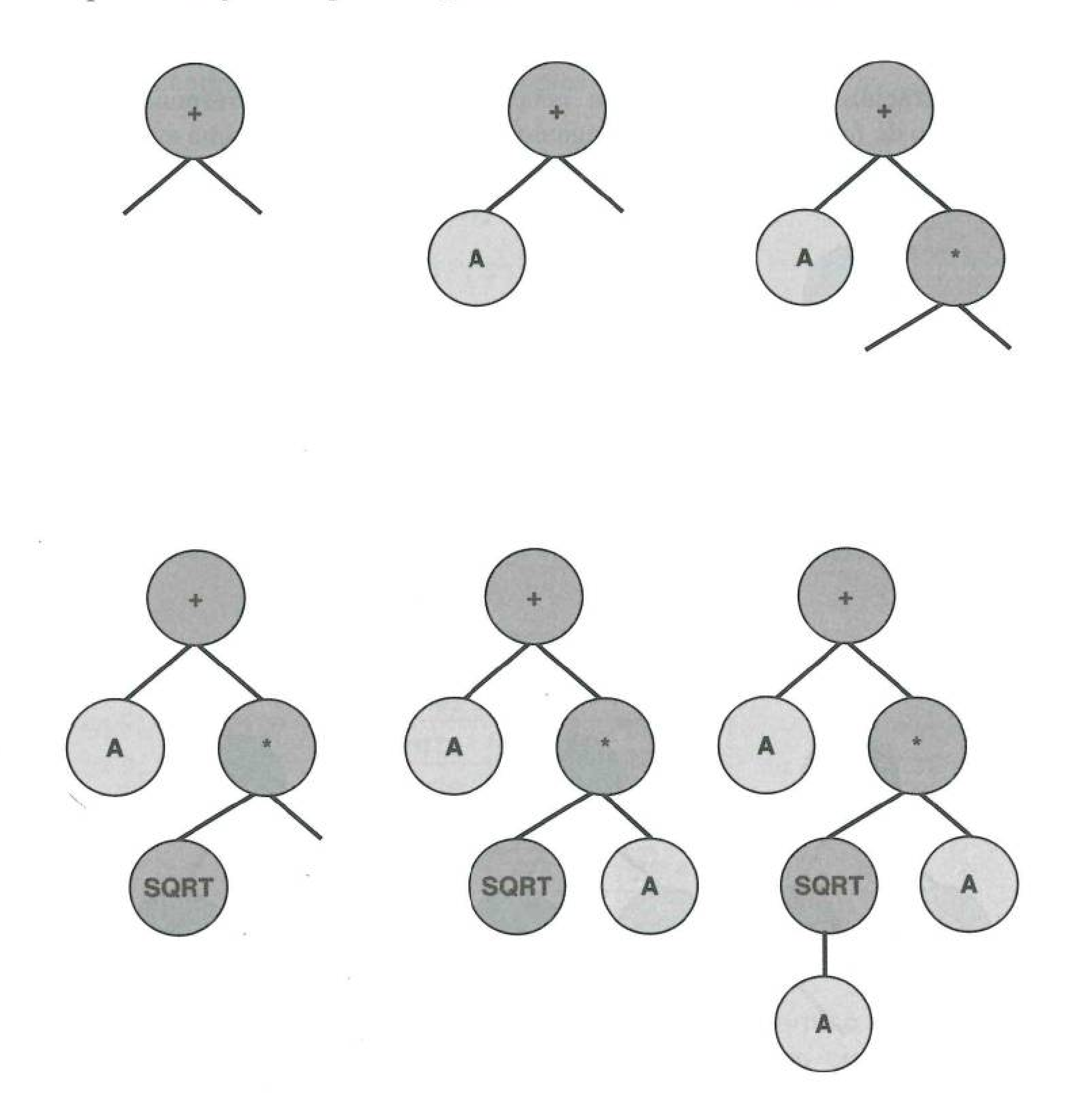
El primer paso consiste en crear los individuos, es decir, árboles válidos de tamaño limitado. Para crear la población inicial podemos utilizar diferentes técnicas de creación de los árboles correspondientes a los programas. Los métodos más utilizados son:

- Inicialización completa
- Inicialización creciente
- Ramped Half-and-Half

- **Inicialización completa:** se van generando los nodos correspondientes al conjunto de funciones hasta que llegamos a la profundidad máxima especificada y en ese punto se generan sólo símbolos terminales.



- **Inicialización creciente**: se van generando los nodos correspondientes al conjunto de funciones y terminales hasta que llegamos a la profundidad máxima especificada y en ese punto se generan sólo símbolos terminales.



También se puede aplicar el método **"Ramped Half-and-Half"** que combina características de los dos métodos anteriores y que se ha comprobado que aporta diversidad a la población. Fijada una profundidad máxima P, se divide la población en P-1 grupos. Cada grupo utiliza una profundidad de árbol diferente

(2,...,P) y la mitad de cada grupo se crea con el método de inicialización creciente y la otra mitad por el método de inicialización completa.

Inicialización completa:

```
funcion inicializacionCompleta(profundidad) {

 si profundidad < maximaProdundidad entonces
   nodo ← aleatorio(conjFunciones)
   para i = 1 hasta número de hijos del nodo hacer
      Hijoi = inicializacionCompleta(profundidad+1 )
 eoc
   nodo ← aleatorio(conjTerminales)
 devolver nodo
}
```

Inicialización creciente:

```
función inicializacionCreciente(profundidad) {

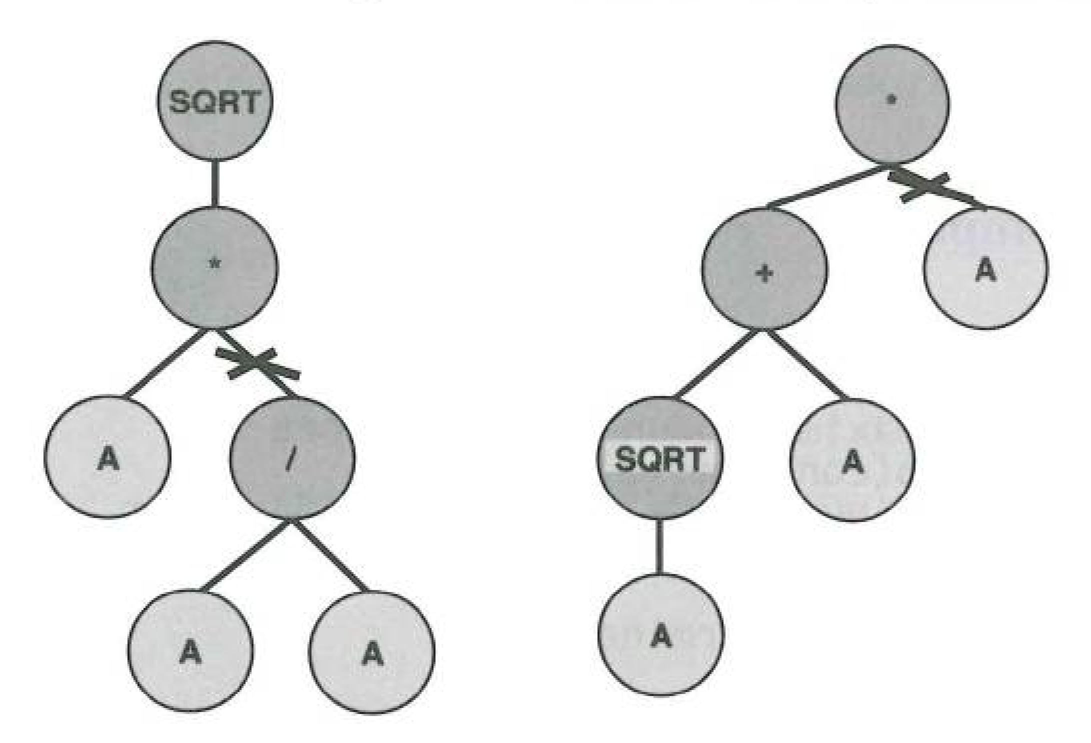
 si profundidad < maximaProdundidad árbol entonces
   nodo ← aleatorio(conjFunciones ⊕ conjTerminales)
   para i = 1 hasta número de hijos del nodo hacer
      Hijoi = inicializacionCreciente(profundidad+1 )
 eoc
   nodo ← aleatorio(conjTerminales)
 devolver nodo
}
```

Una vez inicializada la población y con una configuración de los parámetros del algoritmo se aplican los operadores de selección, cruce y mutación para producir un cierto número de generaciones.

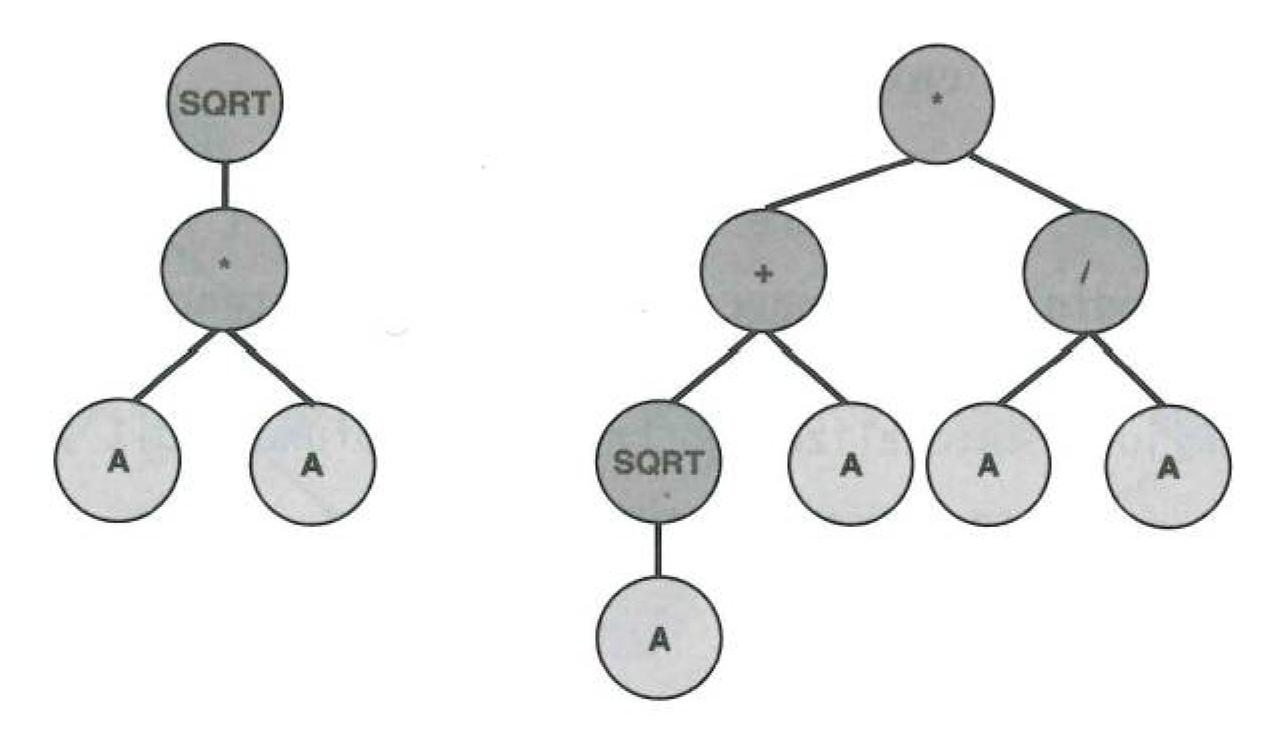
4.3.2 Operadores de cruce

Un cruce consiste en elegir un punto aleatorio de cada padre e intercambiar los subárboles bajo estos puntos para producir dos hijos. El cruce permite que el tamaño del programa aumente o disminuya, por lo que a veces es importante limitar la profundidad de los árboles evitando estructuras que representen programas excesivamente largos.

Si tenemos los siguientes individuos con los puntos de corte:



obtenemos los siguientes hijos:



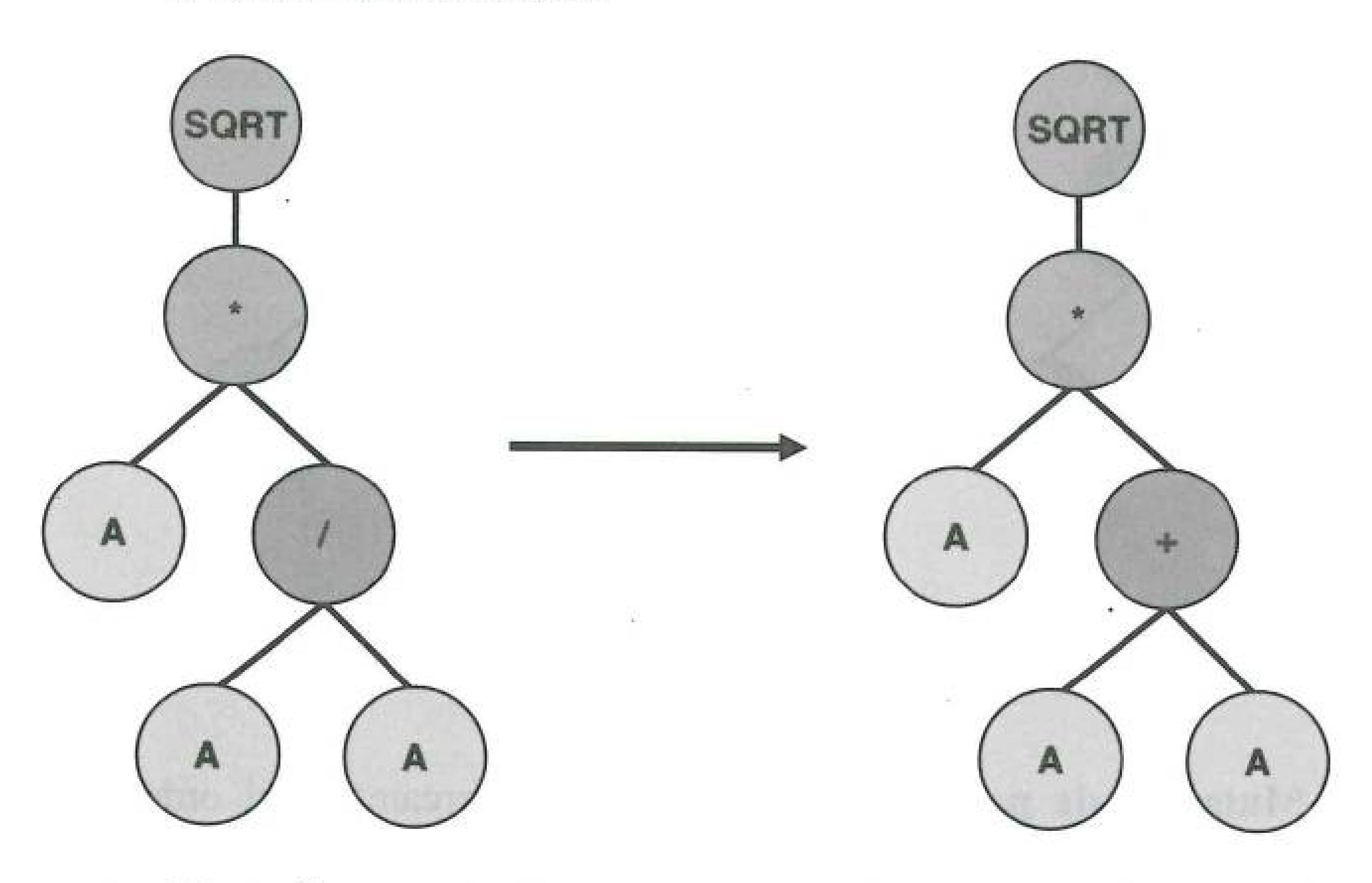
4.3.3 Operadores de mutación

Podemos destacar cuatro métodos diferentes de aplicar mutación a un programa representado en forma de árbol:

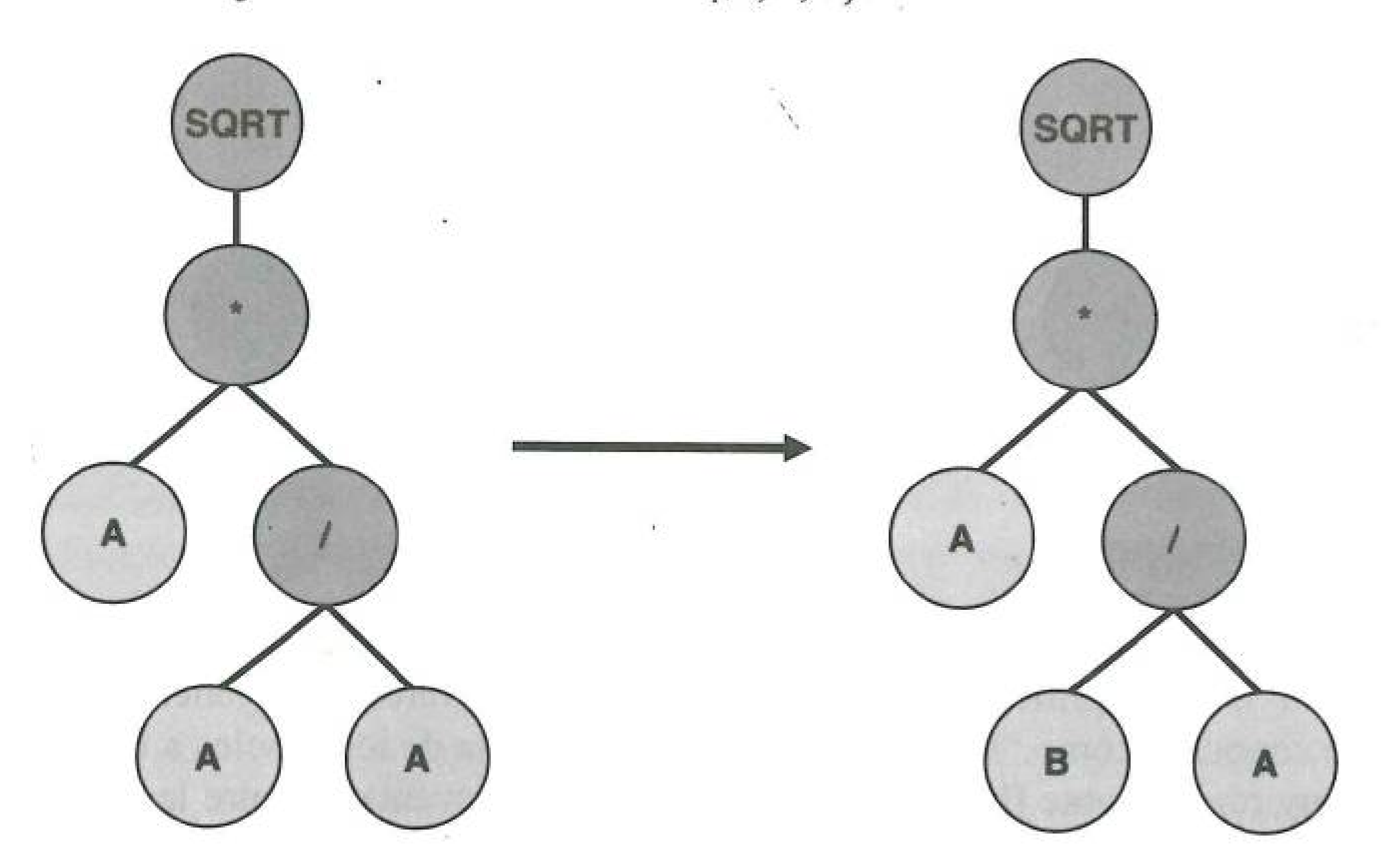
- Mutación funcional
- Mutación terminal
- Mutación de árbol

- Mutación de permutación

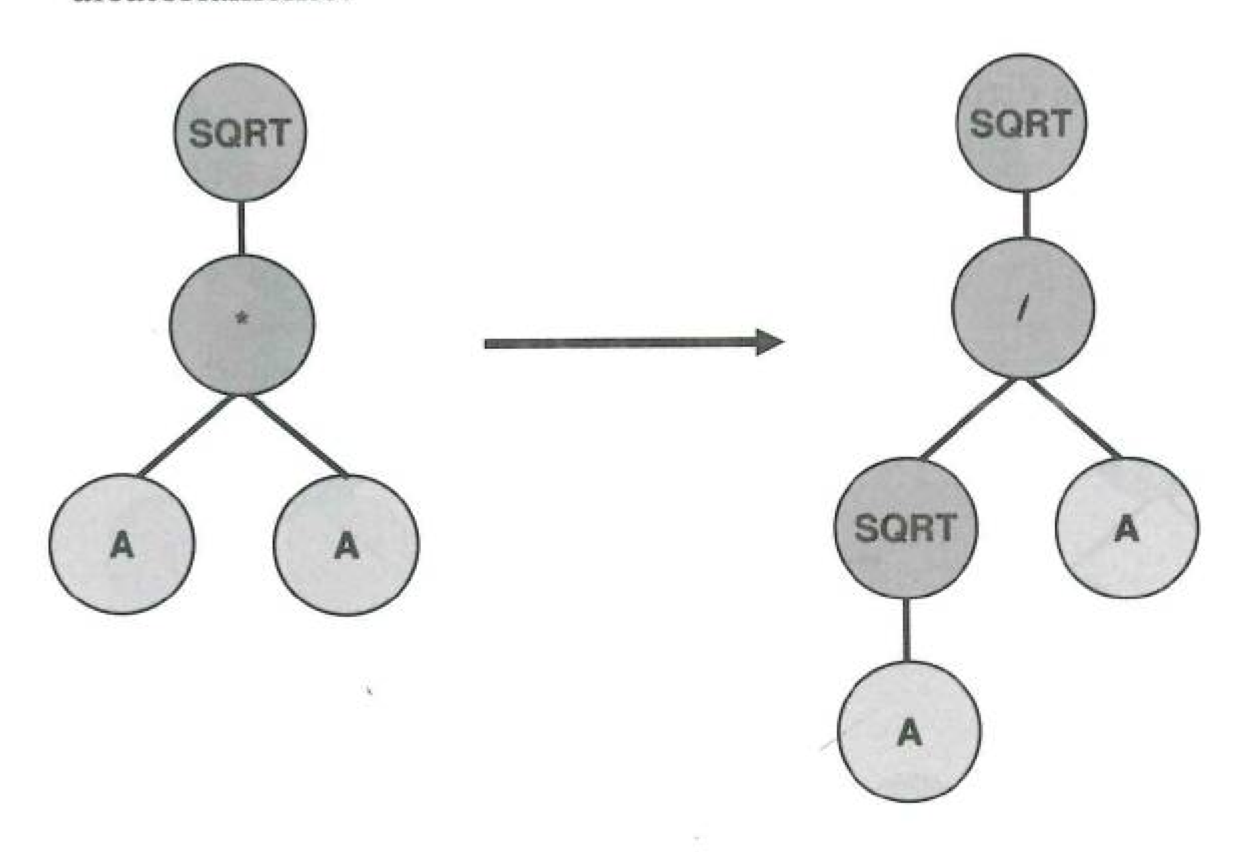
- **Mutación funcional**: consiste en cambiar un operador o función por otro elegido aleatoriamente del conjunto de funciones, respetando la aridad de dicha función.



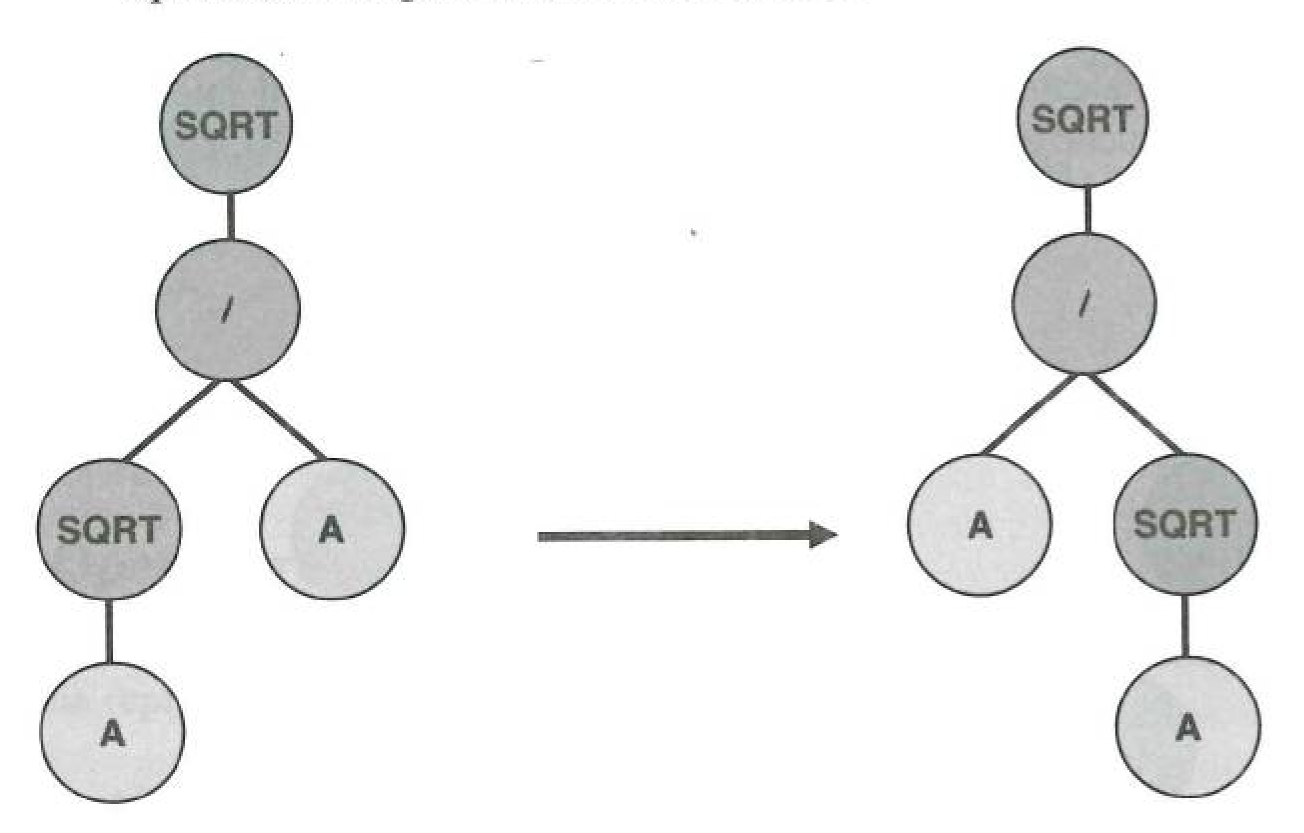
- **Mutación terminal**: consiste en cambiar un símbolo terminal por otro elegido aleatoriamente del conjunto de terminales. En nuestro caso no tiene mucho sentido al considerar un solo terminal A. Si, por ejemplo, el conjunto de terminales fuera {A,B,C}:



- **Mutación de árbol**: consiste en elegir un punto aleatorio del árbol y reemplazar el subárbol bajo este punto por otro nuevo subárbol generado aleatoriamente.



- **Mutación de permutación**: consiste en intercambiar el orden de los operandos o argumentos de una función.



Uno de los principales problemas de la programación genética es el fenómeno conocido como "Bloating", que es la tendencia de los árboles a crecer en tamaño muy rápidamente [KOZ92]. Esto tiene varias desventajas entre las que nos

encontramos con un consumo excesivo de memoria y procesador. Esto se puede resolver de varias formas, aunque la más sencilla consiste en limitar manualmente la profundidad de los árboles.

En la parte de este libro dedicada a los proyectos se describen detalladamente varias aplicaciones de programación genética: identificación de funciones, estrategias de rastreo y juego de marcianos.

Capítulo 5

EXTENSIONES DE LOS ALGORITMOS GENÉTICOS

En este capítulo vamos a presentar variantes del esquema de los algoritmos evolutivos que afectan de distintas formas al esquema general presentado en el capítulo 2. En capítulos anteriores hemos considerado otras variaciones que se referían a los componentes (capítulo 3): operadores, función de adaptación, etc. y a la representación de los individuos (capítulo 4). Los algoritmos que presentamos aquí, aunque de índole muy variada, tienen en común que, en mayor o menor grado, cambian el esquema general de los AEs. Por ejemplo, los algoritmos multiobjetivo, que son aquellos que se utilizan para optimizar simultáneamente más de una función objetivo, pueden formularse definiendo una función de adaptación como una combinación lineal de las diferentes funciones objetivo. Sin embargo, en la mayoría de los casos este esquema no proporciona buenos resultados, y se suelen resolver con otras técnicas que introducen modificaciones al esquema.

Consideramos también en este capítulo versiones paralelas de los AEs. En mayor o menor grado todas ellas suponen alguna modificación del esquema de los AEs, ya que conllevan una distribución de información entre un conjunto de procesadores. Veremos que algunos de los modelos propuestos producen los mismos resultados que una ejecución secuencial, pero en menos tiempo, mientras que otros alteran las características de la población del AE, modificando así su comportamiento.

Los algoritmos meméticos son algoritmos híbridos en los que los AEs se combinan con otras técnicas de búsqueda para explotar al máximo el conocimiento

sobre el problema específico considerado. En general este conocimiento se explota introduciendo en el algoritmo una fase de búsqueda local que mejora la calidad de los individuos.

Las variantes que hemos mencionado hasta aquí, algoritmos multiobjetivo, paralelos, o meméticos, se utilizan desde hace algunas décadas. Revisaremos también en este capítulo otras propuestas más recientes, como las colonias de hormigas, que están teniendo una rápida difusión.

5.1 ALGORITMOS EVOLUTIVOS MULTIOBJETIVO

Los problemas multiobjetivo son muy frecuentes en el mundo real, es decir, a menudo se necesita optimizar simultáneamente varios aspectos de una situación: queremos maximizar la ganancia a la vez que se optimiza el coste de fabricación de un producto, queremos minimizar el tiempo de fabricación y a la vez el coste del material, etc. Formalmente, estos problemas se plantean de la siguiente forma:

$$\begin{cases} Optimizar \quad f_i(x) \qquad (\forall i = 1,\ldots,n) \\ \quad sujeto\ a \qquad\qquad x \in X \subseteq R^m \\ \end{cases}$$

es decir, se trata de optimizar simultáneamente n funciones, cuyas variables pueden estar sujetas a ciertas restricciones.

El concepto de óptimo global en un contexto multiobjetivo ya no es inmediato: ¿qué es mejor, optimizar todo lo posible una de las funciones, a expensas de las otras, una optimización media de todas ellas, etc.?

En un contexto multiobjetivo se suele considerar como óptimo la propuesta originalmente realizada por Francis Ysidro Edgeworth en 1881 y generalizada posteriormente por Vilfredo Pareto en 1896, y que se denomina *óptimo de Pareto*:

Una solución $x \in X$ es un óptimo de Pareto cuando **no** existe ninguna solución mejor en ningún objetivo:

$$\exists y \in X$$

tal que:

- $f_i(y) \leq f_i(x)(\forall i = 1, \cdots, k)$ y
- $f_j(y) < f_j(x)$ para algún j.

Estamos suponiendo una minimización. Se considera que las mejores soluciones estén entre los óptimos de Pareto, de los que en general habrá más de uno. En un contexto multiobjetivo de k funciones, se dice que una solución u domina a otra solución v, y se denota por si $u \preceq v$ si

$$\forall i, 1 \leq i \leq k, f_i(u) \leq f_i(v),$$

y

$$\exists i, 1 \leq i \leq k, f_i(u) < f_i(v).$$

Es decir, suponiendo de nuevo una minimización, u es parcialmente menor a v en todas las funciones, y al menos para alguna de ellas es estrictamente menor (es decir, mejor). Los óptimos de Pareto son las soluciones no dominadas.

Consideremos el siguiente caso. Se quieren optimizar (minimizar) simultáneamente las funciones f_1,f_2,f_3, y f_4. Consideremos la relación de dominancia entre los puntos x, y y z, para los que las funciones toman los siguientes valores:

individuo	f_1	f_2	f_3	f_4
x	3	4	5	4
y	3	3	6	4
z	3	4	5	6

Tanto x como y son óptimos de Pareto, ya que no hay ninguna solución que sea mejor (menor) para todas las funciones. Por el contrario, z no lo es, ya que la solución x es parcialmente mejor que ella para todas las funciones, y estrictamente mejor en la función f_4.

En general, para un problema existe todo un conjunto de óptimos de Pareto, que denotamos por P^*:

$$P^* = \{x \in \Omega \mid \neg \exists\, x' \in \Omega \ \ tal\ que \ \ \vec{f}\,(x') \leq \vec{f}(x)\}$$

Se utiliza también el concepto de frente de Pareto, denotado por PF^*. Se trata de los vectores dados por los valores de las funciones del problema multiobjetivo en los óptimos de Pareto:

$$PF^* = \{\vec{u} = \vec{f} = (f_1(x), \cdots, f_k(x)) \mid x \in P^*\}$$

La primera aplicación de un algoritmo evolutivo a un problema multiobjetivo fue la propuesta de Schaffer [SHA85] y se presentó en 1985 en la Primera Conferencia Internacional de Algoritmos Genéticos, con el nombre de *Vector Evaluated Genetic Algorithm* (VEGA). Desde entonces, se han propuesto y aplicado a numerosos problemas diversos algoritmos evolutivos multiobjetivo. Podemos encontrar varios trabajos dedicados a la revisión y clasificación de dichos algoritmos [FON95, COE01, COE02, KON06]. Las distintas propuestas se suelen clasificar en dos tipos dependiendo de si usan o no el concepto de dominancia de una u otra forma. La propuesta más común de AE multiobjetivo que no utiliza dominancia es la combinación lineal de funciones. Entre las que sí utilizan este concepto existen numerosos esquemas de los que aquí comentaremos algunos de los más utilizados.

5.1.1 Funciones agregativas

La forma más sencilla que se nos puede ocurrir para tratar un problema multiobjetivo será obtener una función única a partir de las distintas funciones a optimizar. Este enfoque se suele denominar funciones agregativas y debido a su sencillez se ha aplicado frecuentemente. De acuerdo con esto, el AE optimiza de la forma habitual una función f obtenida de la siguiente forma:

$$f(x) = w_1 f_1(x) + w_2 f_2(x) + \cdots + w_k f_k(x)$$

donde los w_i son los coeficientes que se utilizan para dar más peso a unas u otras de las funciones a optimizar. Habitualmente se toma el siguiente convenio:

$$\sum_{i=1}^{k} w_i = 1$$

Una de las principales dificultades de utilizar este método es encontrar un conjunto de pesos adecuado.

5.1.2 Aproximaciones que utilizan el concepto de dominancia

Existen decenas de propuestas y variantes de ellas que utilizan de una u otra forma el concepto de dominancia. Como mencionábamos antes, Schaffer [SHA85] propuso el esquema VEGA, que sólo se diferencia de un algoritmo genético clásico en la forma en la que se realiza la selección. En cada generación se producen una serie de subpoblaciones aplicando selección proporcional con respecto a cada una de las funciones a optimizar. Estas subpoblaciones se mezclan para obtener una única población en la que se aplican los operados genéticos de la forma habitual. Las soluciones que proporciona este algoritmo son localmente no dominadas, pero no hay garantía de que lo sean globalmente. El problema de este esquema es que se pueden producir soluciones de mucha calidad respecto a uno de los objetivos pero muy deficientes para el resto.

Fonseca y Fleming [FON93] han propuesto uno de los algoritmos más usados para resolver este tipo de problemas, denominado *Multi-Objective Genetic Algorithm* (MOGA). En este caso, a cada individuo se le asigna un rango dado por el número de individuos de la población que le dominan. A todos los individuos no dominados se les asigna rango 1, y a los dominados el rango que les corresponda. Después, para calcular la adaptación de cada individuo se ordena la población por su rango y se asigna el valor de adaptación mediante alguna función que interpola desde el rango de los mejores hasta el de los peores.

Si no se aplican mecanismos adicionales, la población de los AEs multiobjetivo tiende a agruparse en un número relativamente pequeño de clases de características similares. Por ello, es importante mantener la diversidad de la población y obtener así soluciones uniformemente distribuidas por el frente de Pareto.

Uno de los mecanismos principales que favorece la búsqueda de regiones inexploradas del frente de Pareto es la *compartición de la adaptación*. Este mecanismo consiste en reducir, de forma artificial, el valor de la adaptación de los individuos correspondientes a zonas del espacio con una alta densidad de población. Para ello, se identifican dichas zonas y se aplica un método de penalización. Fonseca y Fleming [FON93] usaron esta idea para penalizar a las soluciones pertenecientes al mismo rango mediante el siguiente proceso:

1. En primer lugar se calcula la distancia Euclídea entre cada par de puntos x e y del conjunto, normalizando el espacio a valores entre 0 y 1:

$$dist(x,y) = \sqrt{\sum_{k=1}^{K} \left(\frac{f_k(x) f_k(y)}{f_k^{max} - f_k^{min}} \right)^2}$$

donde f_k^{max} y f_k^{min} son, respectivamente, los valores máximo y mínimo de la función objetivo f_k alcanzados hasta el momento en la búsqueda.

2. En base a los valores anteriores, se calcula un contador de nicho de cada individuo x de la población:

$$cont_nicho = \sum_{y \in P, r(y,t) = r(x,t)} max \left\{ \frac{\sigma_{comp} - dist(x,y)}{\sigma_{comp}}, 0 \right\}$$

donde σ_{comp} es el tamaño del nicho, un nuevo parámetro a definir. $r(x,t)$ representa el rango del individuo x en la generación t.

3. Después de calcular los contadores de nicho, se ajusta la adaptación de cada individuo de la siguiente forma:

$$f'(x,t) = \frac{f(x,t)}{cont_nicho(x,t)}$$

MOGA fue el primer algoritmo multiobjetivo que utilizó la ordenación basada en Pareto y la compartición de la adaptación.

Otro esquema muy utilizado es el propuesto por Deb y colaboradores [DEB02] y denominado NSGA-II. Este método evita el uso del parámetro de compartición de adaptación usando el concepto de *distancia de saturación*, *ds*. Concretamente los citados autores calculan esta distancia de la siguiente forma:

1. Se ordena la población y se identifican los frentes no dominados F_1, $F_2, \ldots, F_r$. Para cada frente F_i se repiten los pasos 2 y 3 siguientes:

2. Para cada función objetivo f_k se ordenan las soluciones de F_j en orden ascendente. Sea l el tamaño de F_j, y $x_{[i,k]}$ el i-ésimo individuo de la lista ordenada con respecto a la función f_k. Se asigna una distancia de saturación infinita al primer y al último elemento de la lista, $ds_k(x_{[1,k]}) = \infty$ y $ds_k(x_{[l,k]}) = \infty$, y para los restantes:

$$ds_k(x_{[i,k]}) = \frac{f_k(x_{[i+1,k]}) - f_k(x_{[i-1,k]})}{f_k^{max} - f_k^{min}}$$

3. Para encontrar la distancia de saturación total ds(x) del individuo x, se suman las distancias de saturación con respecto a cada objetivo, es decir, $ds(x) = \sum_k ds_k(x)$.

En NSGA-II, una vez calculada esta distancia, se usa para aplicar una técnica de selección denominada operación de selección por torneos de saturación: se seleccionan dos individuos aleatorios *x* e *y*; si ambos pertenecen al mismo frente no dominado, se selecciona el individuo con mayor distancia de saturación. Si no, se selecciona el individuo de menor rango.

Otro concepto muy importante en los AEs multiobjetivos y que en particular está presente en la propuesta de NSGA-II es el elitismo. Se ha comprobado [DEB01] que los esquema de AEs multiobjetivo que incorporan elitismo, en general, proporcionan mejores resultados que los mismos sin elitismo. En los AEs multiobjetivo todas las soluciones no dominadas que se van encontrando se consideran parte de la élite. Sin embargo, debido a la gran cantidad de soluciones de élite de este tipo que se pueden encontrar, la implementación del elitismo no es tan sencilla como en los AEs monoobjetivo. Las estrategias de implementación de elitismo en los AEs multiobjetivos pueden clasificarse en dos grupos: (a) mantener a los individuos de la élite en la población y (b) almacenar las soluciones de la élite en una lista secundaria externa y volver a introducirlas después en la población.

Se han realizado muchas otras propuestas de AEs multiobjetivos, tanto anteriores como posteriores a las descritas, que proponen formas alternativas de implementar los mecanismos que hemos mencionado en esta breve introducción al tema. Muchas de estas técnicas están enfocadas a mejorar la eficiencia computacional.

5.1.3 Ejemplos de aplicación

Los AEs multiobjetivo se han aplicado a muchos problemas prácticos que se pueden encontrar en nuestro entorno: problemas de planificación, de empaquetado, diseño de circuitos, selección de características, diseño de distintos sistemas de control, etc. Mencionamos aquí algunos de ellos sólo a modo de muestra.

Watanabe y colaboradores [WAT03] han desarrollado una versión multiobjetivo del problema del empaquetamiento rectangular, un complejo problema de optimización combinatoria, en el que se trata de colocar una serie de piezas menores rectangulares dentro de rectángulos de hojas de material. El objetivo es encontrar una colocación de los rectángulos de las piezas que requiera la menor área posible de rectángulos de material, es decir, minimizar este área. En la propuesta que los autores hacen en este trabajo, este objetivo se desdobla en dos: (1) minimizar la anchura del área utilizada y (2) minimizar la longitud del área. De esta forma se pueden aplicar técnicas multiobjetivo.

Otra aplicación interesante ha sido propuesta por Watanabe y Sakakibara [SHI06]. Se trata del problema de enrutamiento de vehículos. Es un problema clásico de optimización combinatoria que aparece en mucho sistemas de transporte y distribución. Se trata de minimizar el coste de la ruta desde un depósito central hasta un conjunto de clientes geográficamente dispersos (ciudades, tiendas, etc.). Las rutas se deben diseñar de forma que cada cliente se visite una única vez por un único vehículo, que todas las rutas empiecen y terminen en el depósito central y que la cantidad total de peticiones para cada ruta no exceda la capacidad de un vehículo. El objetivo suele ser minimizar la distancia total recorrida. En la versión multiobjetivo de este trabajo concreto, la versión monoobjetivo se transforma en multiobjetivo añadiendo un nuevo objetivo relacionado con la asignación de vehículos a clientes. Concretamente se optimiza la compacidad del agrupamiento de los clientes asignados a cada vehículo. Después se aplican distintas variantes del algoritmo NSGA-II.

Como tercer ejemplo de aplicación hemos seleccionado un trabajo de Anchor y colaboradores [ANC03] dedicado al problema de la detección de ataques en redes de ordenadores. Los sistemas de detección de intrusos detectan los ataques recogiendo y analizando datos sobre el tráfico en la red. El tráfico de la red está compuesto por una secuencia de paquetes, y un ataque es también una secuencia de paquetes. Las características de los paquetes y las relaciones entre ellos se usan para determinar si una secuencia de paquetes concreta es o no un ataque. Se han utilizado dos enfoques principales para la detección de ataques. En el primero se utiliza conocimiento sobre un ataque, por ejemplo, en forma de patrones, para detectar si una nueva entrada es un ataque. En el segundo enfoque se modelan las secuencias fiables, es decir, que no son ataques, y se consideran ataques aquellas secuencias que se desvían del modelo de las fiables. En el trabajo que describimos aquí se utilizan algoritmos multiobjetivo para aplicar una combinación de ambas estrategias. Una técnica para detectar los ataques es usar un autómata finito que acepte las secuencias que no son ataques y rechazar las secuencias de ataque. Así, se puede utilizar para detectar si una determinada secuencia es o no un ataque. La primera de las funciones de adaptación que se utilizan en la propuesta de este

trabajo mide el porcentaje de símbolos de salida del AF que encajan con un patrón de secuencia de ataque. Las otras funciones miden el porcentaje de símbolos de salida que no encajan con modelos de secuencias correctas, es decir, de cadenas que no son ataques.

5.2 ALGORITMOS EVOLUTIVOS PARALELOS

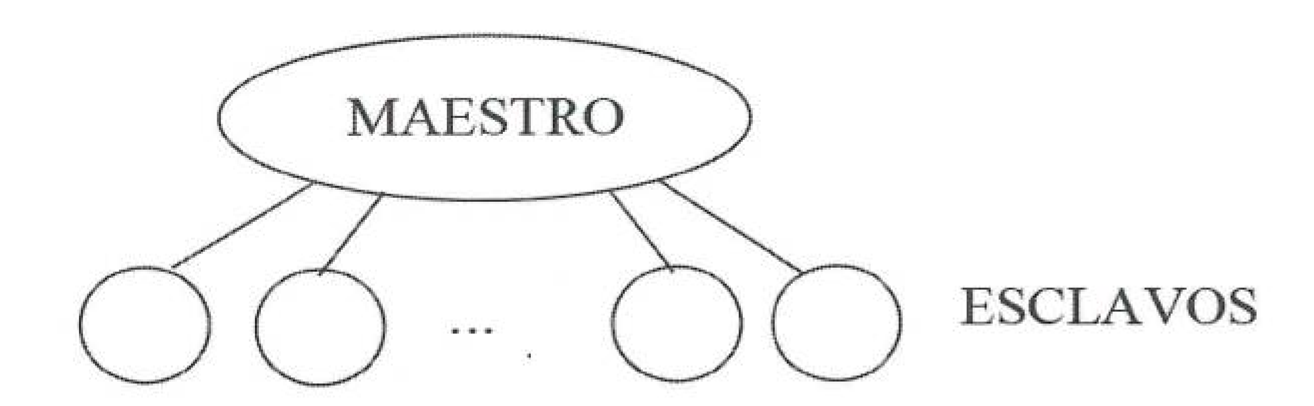
Los algoritmos evolutivos trabajan simultáneamente con un conjunto de individuos o soluciones potenciales, que evolucionan en paralelo. Esta característica sugiere que existe un paralelismo potencial en el desarrollo del algoritmo. Los algoritmos genéticos paralelos son muy fáciles de implementar y pueden aumentar considerablemente el rendimiento. Por este motivo se le ha dedicado una gran atención a la investigación en estas técnicas. Existen algunos trabajos dedicados a la revisión y clasificación de este tipo de algoritmos [CAN97A, CAN00, KON04, ALB02]. La división del problema para su ejecución paralela se puede realizar de distintas formas. En unos casos se mantiene una única población, mientras que en otros la población se divide en múltiples subpoblaciones. Hay esquemas que son más apropiados para unas u otras arquitecturas paralelas de computadores. La clasificación más extendida de este tipo de algoritmos es la siguiente:

- Modelos centralizados o en granja
- Modelos de islas o distribuidos
- Modelos de grano fino o celulares
- Modelos híbridos o jerárquicos

Vamos a revisar aquí brevemente las características de cada uno de estos modelos.

5.2.1 Modelos centralizados o en granja

En este tipo de modelos hay un nodo (procesador) central o "maestro" que coordina el funcionamiento de varios nodos subordinados o "esclavos", como muestra la siguiente figura:



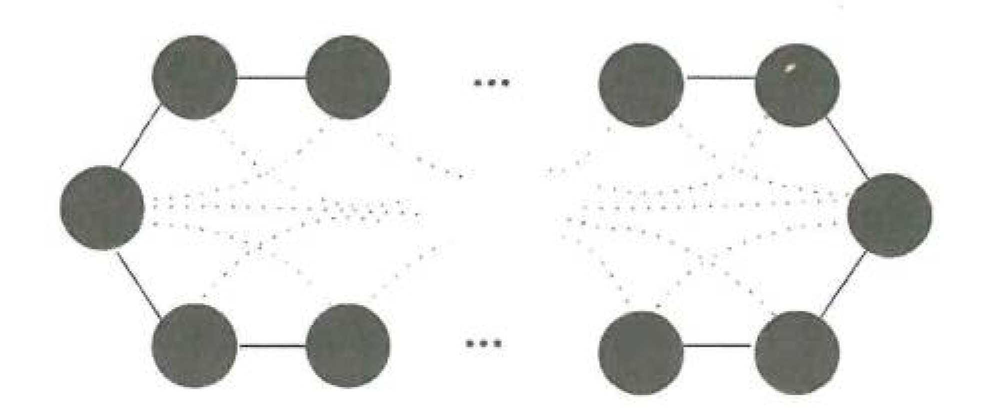
En este tipo de AEs paralelos hay una única población, al igual que en la ejecución secuencial, y como en los AEs secuenciales, cada individuo puede competir y reproducirse con cualquier otro de la población. La operación que se paraleliza en la mayor parte de las implementaciones de este modelo es la evaluación de los individuos, que suele ser la más costosa. Es decir, el nodo maestro se encarga de realizar la selección, el cruce y la mutación, y distribuye la evaluación de la función de adaptación entre los nodos esclavos. Después de evaluar a los individuos que reciben, los nodos esclavos devuelven el resultado al maestro.

Una característica importante de estos algoritmos es que los resultados son los mismos que los de la ejecución secuencial, sólo cambia en tiempo de ejecución, ya que el nodo central ejecuta un AE clásico. En general, el paralelismo que se explota en estos modelos es de grano relativamente fino (la evaluación de uno o un pequeño conjunto de individuos) y con comunicaciones muy frecuentes (en cada generación se envían individuos y se reciben los resultados) por lo que requiere comunicaciones rápidas, siendo apropiado para arquitecturas con memoria compartida. Este modelo presenta algunos inconvenientes que son inherentes al centralismo:

- Necesidad de sincronismo (posibles tiempos de espera).
- Fiabilidad dependiente del proceso central.

5.2.2 Modelos de islas o distribuidos

En estos modelos se tienen una serie de nodos o "islas" que ejecutan simultáneamente el AE. Cada nodo trabaja con una subpoblación de individuos (cada subpoblación se genera con una semilla diferente) e intercambian individuos esporádicamente, como muestra la siguiente figura:



Este intercambio de individuos se denomina "migración". El diseño de este tipo de modelos requiere definir una serie de parámetros relativos a las migraciones:

- Número de generaciones entre intercambios.
- Número de individuos a intercambiar.
- Selección de los individuos a enviar.
- Selección de los individuos a reemplazar por los recibidos.

Estos modelos son probablemente los más populares por dos razones:

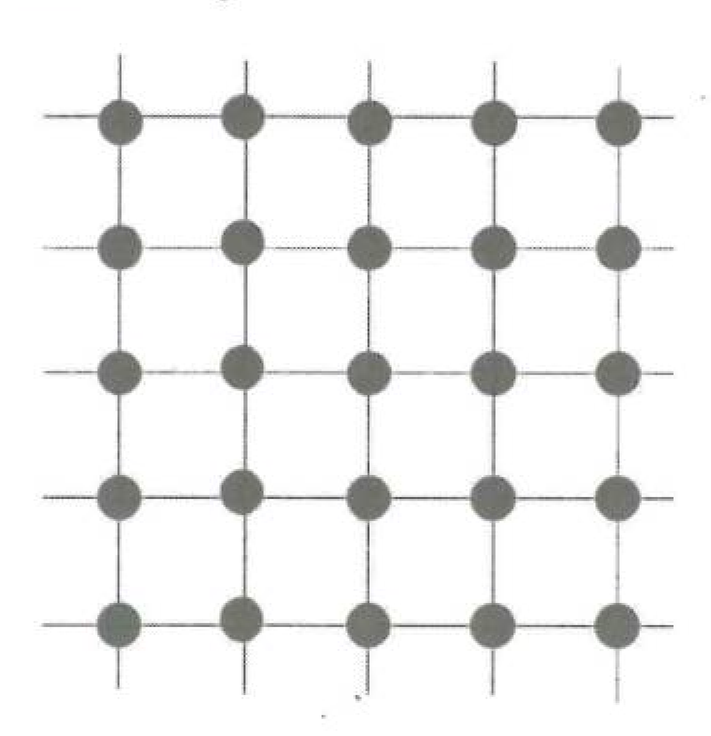
- Su implementación es muy sencilla, ya que utilizan el esquema de un AE clásico, en el que se incluyen primitivas para realizar los intercambios de individuos. Diversos trabajos [ALB01, FER02] han demostrado que el sincronismo entre los nodos no es beneficioso, por lo que pueden utilizar comunicaciones asíncronas, con lo que la implementación resulta aún más sencilla.
- Explotan un paralelismo de grano grueso con comunicaciones esporádicas, por lo que son apropiados para ser implementados en arquitecturas distribuidas de computadores, como redes de PCs, que son claramente las más asequibles y extendidas.

La solución resultante de estos algoritmos es diferente de la que produce la secuencial, ya que las operaciones que se producen en cada subpoblación, que tiene menos individuos que la población global, son diferentes. Al trabajar con poblaciones más pequeñas, estos algoritmos convergen más rápidamente que los

secuenciales. Lógicamente, de esta forma se reduce el tiempo de ejecución. Puede pensarse que esta evolución en poblaciones más pequeñas puede conllevar problemas de falta de diversidad para determinadas aplicaciones. Sin embargo, el análisis realizado en algunos trabajos [CAN97B, CAN97C] ha demostrado que no ocurre así, ya que el intercambio de individuos procedentes de distintas poblaciones favorece la diversidad de forma notable, por lo que estos algoritmos, además de ser más rápidos, frecuentemente alcanzan soluciones mejores que las ejecuciones secuenciales.

5.2.3 Modelos de grano fino o celulares

Estos modelos trabajan con una única población estructurada espacialmente, como muestra la figura:



Habitualmente, la estructura de la población es una rejilla rectangular de dos dimensiones, con un individuo en cada punto de la rejilla. Teóricamente habrá un nodo por individuo, de forma que el cálculo de la adaptación se realiza simultáneamente para todos los individuos.

En este modelo la selección y la reproducción se restringen a la vecindad de cada individuo. Como las vecindades se solapan, se produce una tendencia a que los mejores individuos se expandan por toda la población. El aislamiento de los individuos produce una gran diversidad y baja presión selectiva. Este tipo de modelos se denominan a veces celulares por analogía con un tipo de autómatas celulares con reglas de transición estocásticas [WHI93].

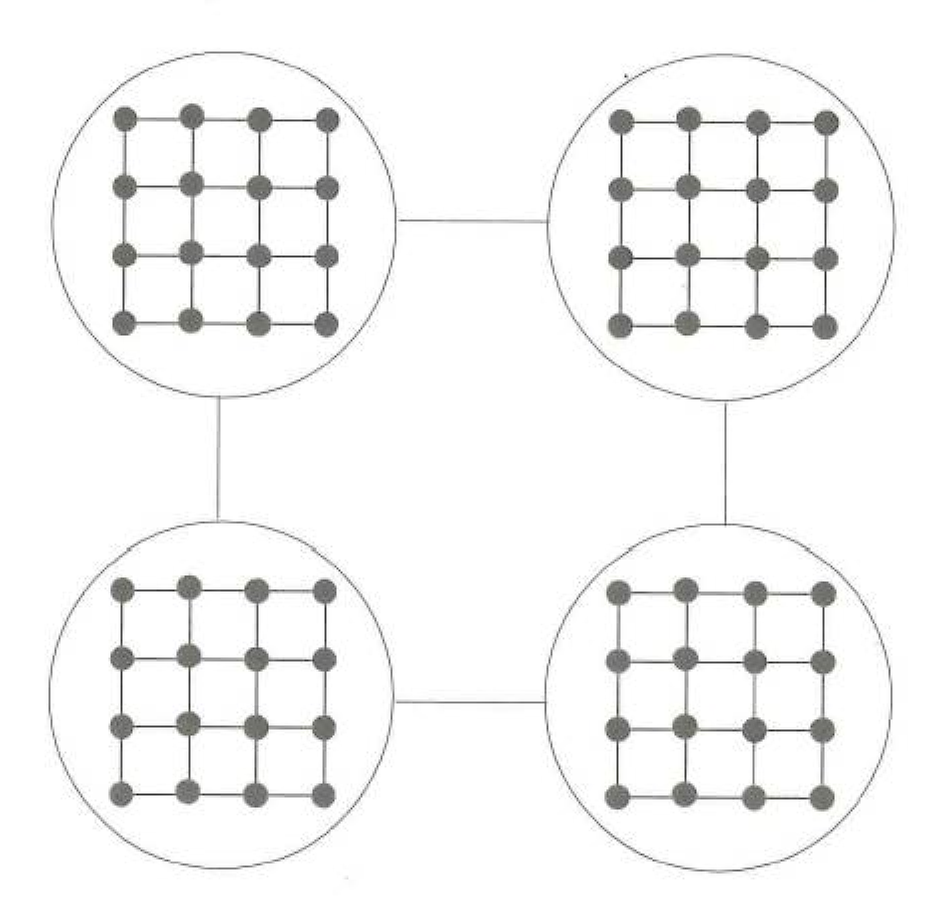
Puesto que el tipo de paralelismo que explotan estos modelos es de grano fino, los computadores masivamente paralelos son apropiados para su

implementación. Sin embargo, también ha habido implementaciones eficientes en arquitecturas distribuidas [LAR08].

5.2.4 Modelos híbridos

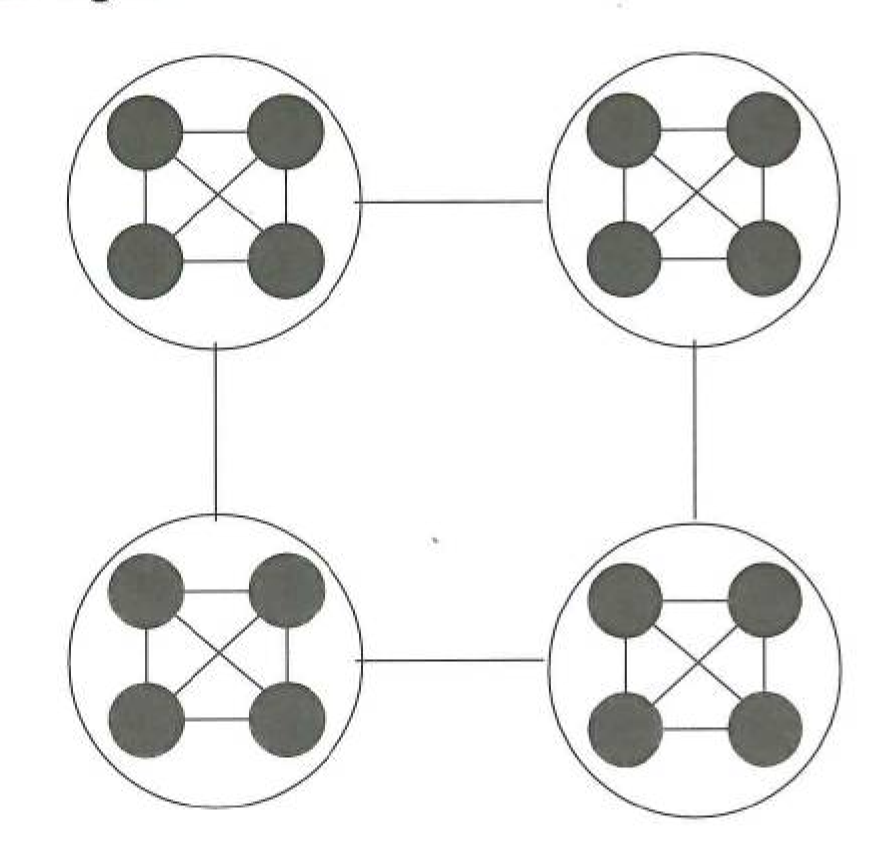
También se han realizado trabajos desarrollando AEs paralelos en los que se combinan dos de los métodos descritos anteriormente. Al realizar esta combinación se forma una jerarquía entre los nodos.

En el nivel más alto están los algoritmos distribuidos, y en el nivel inferior suele estar alguno de los modelos de grano fino. Por ejemplo, la siguiente figura muestra un AE jerárquico propuesto por Gruau [GRU94] que combina un modelo de islas en el nivel más alto, con un modelo celular en el nivel más bajo.



Análogamente, el modelo de islas se puede combinar con un modelo global a bajo nivel, como han hecho Bianchini y Brown [BIA93]:

También se puede utilizar el modelo de islas a ambos niveles, como muestra la siguiente figura:



Herrera y colaboradores [HER99] han realizado un detallado estudio de modelos jerárquicos distribuidos, encontrando que las combinaciones apropiadas de modelos de distintos tipos pueden mejorar los resultados del algoritmo, aunque es necesario ajustar adecuadamente el conjunto de parámetros que surgen en estos modelos.

5.3 ALGORITMOS MEMÉTICOS

Los algoritmos meméticos (AMs) [MOS99, MOS03, KRA05] combinan la mejora evolutiva de una población con la búsqueda local. Se trata de algoritmos inspirados en los modelos de adaptación de los sistemas naturales que combinan la adaptación evolutiva de la población con el aprendizaje individual que tiene lugar durante su período de vida. El nombre de memético proviene del término inglés *meme* introducido por Richard Dawkins [DAW76] para representar una unidad de evolución cultural. En el contexto de la optimización evolutiva un meme representa una estrategia de aprendizaje o de desarrollo.

La estructura de un AM es similar a la de un algoritmo evolutivo. La principal diferencia es que se introduce un nuevo operador que se encarga de la optimización local. Un posible esquema de dicho operador es el siguiente:

```
funcion opt_local(TIndividuo x, TTipo_operador operador)
{
   TIndividuo y; // nuevo individuo

   mientras no condición_terminación_local hacer {
      y = aplicar(operador, x);
      si adaptación(y) > adaptacion(x) entonces
         x = y ;
   }
   devolver x;
}
```

Podemos ver que se trata de un proceso iterativo en el que se aplica el operador indicado en los parámetros de la función y que conserva los cambios que mejoran la adaptación del individuo. El proceso se repite hasta que se alcanza la condición de terminación especificada (cierto número de iteraciones predefinido, convergencia por producirse cierto número de iteraciones sin mejora, haber alcanzado un valor de adaptación especificado, etc.).

En muchos casos los AMs toman la forma de un AE con un operador de búsqueda local que se aplica a todos los individuos de una nueva generación antes de aplicar el operador de selección. Sin embargo, esta es una visión restrictiva de los AMs ya que existen muchas otras formas de combinar los AEs con búsqueda local. De forma más general, podemos considerar que un AM es un AE que incluye una o más fases de búsqueda local en su ciclo evolutivo. Por ejemplo, se pueden utilizar al crear la población inicial o después de cualquier operador genético.

Otra característica que suelen presentan los AMs es un proceso de renovación de la población o reinicialización en caso de convergencia.

```
funcion Algoritmo_Genético()
{
   TPoblacion pob;     // población
   TParametros parámetros// tamaño población

   obtener_parametros(parametros);
   pob = poblacion_inicial();
   evaluacion(pob, tam_pob, pos_mejor, sumadaptacion);

   // bucle de evolución
   mientras no se alcanza la condición de terminación
     hacer{
       selección(pob, parámetros);
       reproduccion(pob, parámetros);
       evaluacion(pob, parámetros, pos_mejor,
     sumadaptacion);
       si convergencia(pob) entonces
          pob = reiniciar_poblacion(pob);
   }
   devolver pob[pos_mejor];
}
```

Cuando la diversidad de la población se reduce por debajo de un umbral predefinido se considera que el algoritmo ha entrado en una situación de convergencia, desde la que la evolución apenas tendrá efecto. La medida de la diversidad puede medirse de distintas formas (diversidad de los valores de adaptación, de los genotipos, etc.). En esta situación la población se reinicializa, conservando parte de ella y generando nuevos individuos hasta completar su tamaño.

```
funcion reiniciar_poblacion(TPoblacion pob, entero tam_pob,
                                      entero num_conserva)
{
   TPoblacion nueva_pob; // nueva población
   TIndividuo indiv;
   entero i;

   para cada i desde 0 hasta num_conserva hacer{
      nueva_pob[i] = siguiente_mejor(pob, i);
   }
   para cada i desde num_conserva hasta tam_pob hacer{
      nueva_pob[i] = genera_indiv(lcrom);
   }
   devolver nueva_pob;
}
```

Los AMs se han aplicado a numerosos problemas. Krasnogor y Smith [KRA05] presentan un estudio de su aplicación a algunos de ellos: el problema del viajante de comercio, el problema de la predicción de la estructura de las proteínas y el problema del coloreado de grafos.

5.4 NUEVAS TENDENCIAS

En los años más recientes se han hecho diversas propuestas de nuevos algoritmos inspirados en la naturaleza. Algunas de estas propuestas se han desarrollado muy rápidamente, aplicándose con éxito en muy poco tiempo a numerosos problemas, y recibiendo la atención de numerosos investigadores. Hacemos en este capítulo una breve presentación de estas propuestas, concretamente, de los algoritmos de colonias de hormigas, de la evolución diferencial, de los algoritmos de estimación de distribuciones y de la evolución gramatical. Esta presentación no pretende ser una guía para su aplicación, sino simplemente un repaso de las ideas principales que subyacen a cada uno de ellos, y que pueden animar al lector a profundizar en su conocimiento utilizando las referencias que se aportan para cada uno de ellos.

5.4.1 Inteligencia colectiva y Algoritmos de Colonias de Hormigas

La inteligencia colectiva o de enjambres (*swarm intelligence*) es el término con el que se designa a los sistemas basados en el comportamiento de colectividades, como insectos o partículas, en los que un conjunto de agentes de comportamiento muy simple dan lugar a un funcionamiento global complejo. Una población de agentes interaccionan con su entorno y entre ellos mediante un conjunto de reglas muy simples. En estos sistemas el control está completamente distribuido entre los agentes, que son autónomos y toman decisiones basadas en información local.

Dentro de los algoritmos de optimización de este tipo, uno de los más destacados es el de optimización por Colonias de Hormigas (ACO)[DOR96]. Se trata de un algoritmo inspirado en el comportamiento de las colonias de hormigas, que se ha aplicado a numerosos problemas de planificación y optimización de rutas, como el viajante de comercio [DOR97] y el enrutamiento de vehículos [RIZ07]. Los elementos principales de estos algoritmos son las hormigas artificiales, que son agentes simples que de forma individual e iterativa construyen soluciones al problema, previamente modelado en forma de grafo. Las hormigas exploran el grafo visitando nodos que estén conectados por enlaces. Una solución al problema es una secuencia ordenada de nodos. El proceso de búsqueda se ejecuta en paralelo sobre varios caminos de construcción. El proceso de construcción de cada uno de estos caminos está guiado por la información contenida en una estructura dinámica sobre la efectividad de los resultados previos del camino.

Este mecanismo está inspirado en el proceso de construcción de rastros de feromonas de las hormigas. Las soluciones parciales al problema pueden verse como estados, de manera que en la iteración k del algoritmo cada hormiga se mueve desde el estado $x_k^{(i)}$ al estado $x_{k+1}^{(j)}$, extendiendo la solución parcial que llegaba al nodo i con el nodo j.

En el primer sistema de hormigas, propuesto por Dorigo y colaboradores [DOR96], el algoritmo se divide en dos etapas:

- Construcción de una solución.
- Actualización del rastro de feromonas.

En este algoritmo cada hormiga construye una solución. Una hormiga que se encuentra en un determinado estado calcula el conjunto de extensiones válidas desde el nodo correspondiente a dicho estado. Después selecciona el movimiento que extiende el estado teniendo en cuenta los siguientes valores:

- El atractivo del movimiento η_{ij}, calculado mediante alguna heurística que indique el interés de ese movimiento a priori.

- El nivel del rastro de feromonas del movimiento τ_{ij}, que indica la utilidad que ha tenido en el pasado ese movimiento.

Dados estos dos valores, la hormiga decide visitar el nodo j desde el nodo i en función de la siguiente probabilidad:

$$p_{ij} = \begin{cases} \dfrac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{h\in\Omega}[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}, & si\ j \in \Omega \\ 0, & en\ otro\ caso \end{cases}$$

donde Ω es el conjunto de nodos que pueden visitarse desde i. Los parámetros α y β ponderan la influencia del rastro y el atractivo. Después de obtener la solución, el rastro de feromonas se actualiza. En primer lugar, las feromonas se evaporan en todos los enlaces, lo que hace que se vayan olvidando las soluciones de baja calidad. Después todas las hormigas depositan feromonas en los enlaces que son parte de las soluciones que acaban de calcular. Existen distintas variantes del algoritmo original, que se diferencian básicamente en la regla de actualización de las feromonas.

5.4.2 Evolución diferencial

La evolución diferencial (ED) [PRI05] es un modelo evolutivo propuesto por Storm y Price [STO97] para tratar problemas que requieren una optimización global en espacios continuos. Normalmente, el objetivo es optimizar ciertas propiedades del sistema eligiendo de la forma más adecuada los parámetros de dicho sistema. Los parámetros del sistema se suelen representar en forma de vector. Estos algoritmos trabajan con una población de vectores, que son perturbados con otros vectores, los de diferencias, y hacen énfasis en la mutación, utilizando un operador de cruce/recombinación posterior a la mutación.

Las características específicas de la ED son las siguientes:

- La población consiste en NP vectores D-dimensionales de parámetros.

$$x_{i,0} = [x_{1,i,0}, \cdots, x_{D,i,0}] \ \ con \ 1 \leq i \leq NP$$

La población inicial se genera aleatoriamente, con cada valor dentro de los límites inferior y superior del parámetro correspondiente.

- En cada generación g se genera una población de prueba $P_{u,g}$ formada por NP vectores D-dimensionales $v_{i,g}$ = $[v_{1,i,g},\ldots v_{D,i,g}]$ mediante las operaciones de mutación y recombinación aplicadas a la población en curso $P_{x,g}$.

Mutación:

Genera nuevos vectores de parámetros añadiendo la diferencia ponderada entre dos vectores a un tercer vector. Para cada vector de la población en curso $x_{i,g}$, que denominamos vector de referencia, se genera un vector mutado $v_{i,g}$ de la siguiente forma:

$$v_{i,g+1} = x_{r_1,g} + F.(x_{r_2,g} - x_{r_3,g})$$

con índices aleatorios r_1, r_2, $r_3 \in \{1,\ldots,NP\}$ enteros y diferentes entre sí, y con $F>0$. Los índices aleatorios también tienen que ser diferentes del índice en curso i. F es una constante real que toma valores entre 0 y 2 y controla la amplificación de la variación diferencial.

Cruce:

Los parámetros del vector mutado se mezclan con los parámetros del vector de referencia, para dar lugar al vector de prueba. Concretamente, se construye el siguiente vector de prueba:

$$u_{i,g+1} = (u_{1,i,g+1}, \cdots, u_{D,i,g+1})$$

Donde:

$$u_{j,i,g+1} = \begin{cases} v_{j,i,g+1} & si\ (randb(j) \leq CR)\ o\ j = rnbr(i) \\ x_{j,i,g} & si\ (randb(j) > CR)\ o\ j \neq rnbr(i) \end{cases}$$

$$j = 1,2,\cdots,D$$

Siendo *randb(j)* la j-ésima evaluación de un generador de números aleatorios con salida entre 0 y 1. *CR* es una constante de cruce entre 0 y 1 que especifica el usuario. *rnbr(i)* es un índice aleatorio elegido entre 1,...,*D*, que asegura que $u_{j,i,g+1}$ tome al menos un parámetro de $v_{j,i,g+1}$.

Selección:

Cada vector de la población tiene que servir una vez como vector de referencia de manera que en cada generación tienen lugar *NP* competiciones. Para decidir si un vector de prueba pasa a ser miembro de la siguiente generación $g + 1$, se le compara con el vector de referencia $x_{i,g}$. Si el vector de prueba da lugar a un valor mejor de la función objetivo, se le incluye en la siguiente generación. Sino, se mantiene a $x_{i,g}$.

Este esquema admite variantes que pueden afectar a la elección del vector a ser mutado (aleatorio, el mejor, etc.), al número de vectores de diferencia considerados en la mutación y en el tipo de cruce utilizado.

5.4.3 Algoritmos de Estimación de Distribuciones

Los Algoritmos de Estimación de Distribuciones (EDAs) [MUH96, LAR01, LAR03] son una variante de algoritmo evolutivo en la que las operaciones de cruce y mutación se sustituyen por operadores que aprenden y muestrean a

partir de la distribución de probabilidad de los mejores individuos de la población en cada iteración del algoritmo. Así una ventaja de estos algoritmos frente a los AEs clásicos es que reducen el número de parámetros que se requieren, uno de los puntos más críticos para el buen funcionamiento de estos algoritmos. En esta propuesta, el cuello de botella del cómputo es la estimación de la distribución de probabilidad conjunta de los individuos seleccionados.

De forma esquemática, estos algoritmos funcionan de la siguiente forma. Comienzan generando M individuos al azar, por ejemplo a partir de una distribución uniforme para cada variable. Estos individuos tienen una componente o gen por cada variable del problema. Por ejemplo, consideramos el siguiente problema, presentado por Larrañaga y colaboradores [LAR03] como ejemplo, en el que se busca el máximo de la función:

$$h(x) = \sum_{i=1}^{6} x_i,\ con\ x_i = 0,1$$

Tenemos entonces 6 variables en el problema y algunos individuos de la población generados usando la siguiente distribución de probabilidad:

$$p_0(x) = \prod_{i=1}^{6} p_0(x_i),\ donde\ p_0(X_i = 1) = 0.5\ para\ i = 1 \cdots, 6$$

son los siguientes:

	X_1	X_2	X_3	X_4	X_5	X_6	h(x)
1	1	0	1	0	1	0	3
2	0	1	0	0	1	0	2
...		...			...		...

Los M individuos constituyen la población inicial, D_0 y se evalúan de la forma habitual en los algoritmos genéticos. A continuación el algoritmo consiste en un bucle que se repite hasta que se cumple la condición de parada que se haya establecido. Los pasos de este bucle son los siguientes:

- Seleccionar un número N ($N \leq M$) de individuos de la población de individuos en curso, normalmente aquellos con mejores valores de función objetivo.

- Estimar la distribución de probabilidad de los individuos seleccionados.

- Muestrear de la distribución de probabilidad estimada, para obtener M individuos que constituyen la nueva población.

La dificultad principal de los EDAs es la forma de estimar la distribución de probabilidad $p_l(x)$. El cálculo directo de la distribución de probabilidad conjunta, incluso para números no muy grandes de variables, es impracticable. La forma de tratar este problema es trabajar con un modelo simplificado en el que la distribución de probabilidad se factoriza. Existen distintas variantes de EDAs en función de la estimación de la distribución de probabilidad que utilizan. Algunas de ellas suponen independencia entre las variables, otras consideran dependencias entre pares de variables y otras entre múltiples variables.

5.4.4 Evolución gramatical

La evolución gramatical, propuesta por C. Ryan y colaboradores [RYA98], es una variante relativamente reciente de la programación genética. Como en el caso de la programación genética, el objetivo es encontrar programas representados de alguna forma por una expresión en forma de árbol, para resolver cualquier instancia de un problema genérico.

En la programación genética de Koza [KOZ02] cualquier función del conjunto de operadores puede tener como argumento a cualquier otra función. Esto causa una explosión combinatoria de posibilidades que lleva a un espacio de búsqueda difícil de abordar sin imponer restricciones adicionales, por ejemplo, a la profundidad del árbol o al número de nodos. La evolución gramatical se diferencia de la programación genética en este punto. La idea es incorporar conocimiento específico del problema en forma de gramática especificada por el usuario. Esta gramática se suele expresar en "forma normal de Backus" (BNF usualmente). Se trata de una notación para representar la gramática de un lenguaje en forma de reglas de producción. En esta notación una gramática se representa por una tupla $\{N,T,P,S\}$, donde N es un conjunto de no terminales, T es un conjunto de terminales, P es un conjunto de reglas de producción que hace corresponder elementos de N a T, y S es el símbolo inicial que pertenece a N. Un ejemplo de gramática para representar una expresión simple es el siguiente:

$$N = \{expr, op, pre-op\}$$
$$T = \{Seno, Cos, Tan, Log, +, -, /, *, X, ()\}$$
$$S = \langle expr \rangle$$

Y el conjunto de reglas de producción se puede representar por:

$$\begin{array}{lcll}(1)\langle expr\rangle & ::= & \langle expr\rangle\langle op\rangle\langle expr\rangle & (A)\\ & | & (\langle expr\rangle\langle op\rangle\langle expr\rangle) & (B)\\ & | & \langle pre-op\rangle(\langle expr\rangle) & (C)\\ & | & \langle var\rangle & (D)\end{array}$$

$$\begin{array}{lcll}(2)\langle op\rangle & ::= & + & (A)\\ & | & - & (B)\\ & | & / & (C)\\ & | & * & (D)\end{array}$$

$$\begin{array}{lcll}(3)\langle pre-op\rangle & ::= & Seno & (A)\\ & | & Cos & (B)\\ & | & Tan & (C)\\ & | & Log & (D)\end{array}$$

$$(4)\langle var\rangle \quad ::= \quad X$$

Otra diferencia con la programación genética es que en la evolución gramatical los fenotipos y genotipos de los individuos no coinciden. En la evolución gramatical, los individuos son listas ordenadas de enteros, de longitud variable, que se utilizan en la selección de la regla de la gramática especificada. A partir de estos genotipos se construyen fenotipos del mismo tipo de los individuos de la programación genética, y que se evalúan de la misma forma. Consideremos el siguiente ejemplo de genotipo:

220	203	17	3	109	215	104	30

Consideremos la regla (1) de la gramática de ejemplo que hemos descrito. Esta regla tiene cuatro posibilidades. Entonces para generar el fenotipo correspondiente al genotipo generamos un número aleatorio entre 1 y 4, usando 220 como semilla del generador de números aleatorios. Supongamos que se ha seleccionado la regla (C) (<pre-op> (< expr>)). Entonces el siguiente gen, 203, se utiliza como semilla para seleccionar un elemento <pre-op>, y así sucesivamente. Es posible que mientras estamos construyendo el genotipo lleguemos al final de la secuencia de genes. En ese caso consideramos al individuo circular y volvemos a utilizar los genes.

Al tener los individuos la misma representación que utilizan los algoritmos genéticos, se pueden utilizar los mismos operadores, facilitando de esta forma la programación de estos algoritmos. Los operadores clásicos se pueden combinar con otros específicos de la evolución gramatical, como la *duplicación* y la *poda*, propuestos por Ryan y colaboradores [RYA98]. El operador de duplicación selecciona múltiples genes y los duplica. Las copias se colocan a continuación del último gen del genotipo. En la evolución gramatical a veces los individuos no utilizan todos sus genes. El operador de poda se utiliza para reducir el número de genes no utilizados. Este operador se aplica con cierta probabilidad, dada por un parámetro, a los individuos que no utilizan todos sus genes. El operador elimina todos los genes que no se utilizan al hacer la correspondencia entre el genotipo y el fenotipo.

PROYECTOS

Proyecto 1: Optimización de Funciones (A)

Proyecto 2: Optimización de Funciones (B)

Proyecto 3: Búsqueda de rutas de metro

Proyecto 4: Planificación de horarios

Proyecto 5: Cortado de patrones

Proyecto 6: Control de tráfico aéreo

Proyecto 7: Resolución del sudoku

Proyecto 8: Identificación de funciones

Proyecto 9: Generación de estrategias de rastreo

Proyecto 10: Invasores del espacio

Proyecto 1

OPTIMIZACIÓN DE FUNCIONES (A)

En este proyecto implementaremos un algoritmo genético clásico para hallar el máximo (o el mínimo) de una función. Comenzaremos con la búsqueda del máximo de una serie de funciones, y nos ocuparemos después de la búsqueda del mínimo, que requiere un pequeño ajuste del algoritmo. Finalmente, veremos cómo buscar el óptimo de funciones de varias variables. En este proyecto se utiliza un algoritmo genético en su forma básica, por lo que resulta ideal para el entrenamiento en la utilización de las técnicas de los algoritmos evolutivos en general y de los genéticos en particular.

En el Apéndice B se muestra el código C++ correspondiente a este proyecto.

1. MAXIMIZACION DE FUNCIONES

Buscamos el máximo de las siguientes funciones:

- $f = sen^6(5\pi x)e^{-2ln(2)(\frac{x-0.1}{0.8})^2} : x \in [0.0, 1.0]$

que presenta un máximo (1.0) en 0.1.

- $f(x) = 20 + e - 20e^{-0.2|x|} - e^{\cos 2\pi x} : x \in [0, 32]$

que presenta un máximo (22.3136) en 31.5005.

- $f = x + | sen(32\pi x) | : x \in [0,1]$

que presenta un máximo (1.98442447) en 0.98447395.

- $f(x) = 0.5 - \dfrac{sen^2(x) - 0.5}{1 + 0.001x^4} : x \in [0, 10]$

que presenta un máximo (1.0) en $x = 0.0$.

Para buscar el máximo de una función implementaremos en cualquier lenguaje de programación los algoritmos presentados en el capítulo 2 dedicado a los algoritmos genéticos. Es decir, las opciones de diseño son las siguientes:

- **Representación de los Individuos**: se representan mediante cadenas binarias que se corresponden con los puntos del espacio de búsqueda.
- **Función de evaluación**: es el resultado de evaluar la función considerada en el punto que resulta de la decodificación del individuo.
- **Selección**: por ruleta.
- **Operador de Cruce**: operador monopunto que selecciona un punto de cruce aleatoriamente y se intercambian los segmentos resultantes.
- **Operador de Mutación**: aleatoria bit a bit que cambia el valor del punto considerado para mutación.

Resulta ilustrativo representar gráficamente la función para comprobar que el óptimo se corresponde con el valor encontrado por el algoritmo evolutivo. También es interesante representar los valores medio y máximo de la función de aptitud a lo largo de las generaciones, para estudiar su evolución.

Para realizar un **estudio de los parámetros** más adecuados para el algoritmo implementaremos un sistema que permita variar los parámetros interactivamente. Los parámetros que se especifican son:

- Valor del error permitido para la discretización del intervalo
- Tamaño de la población
- Número límite de iteraciones del algoritmo

- Porcentaje de cruces
- Porcentaje de mutaciones

El desarrollo del proyecto debe incluir un estudio de los resultados obtenidos con distintos parámetros. Para ello fijamos todos los parámetros menos uno que vamos variando para realizar el estudio.

La introducción del mecanismo de *elitismo* tanto en este apartado como en los siguientes mejora notablemente el comportamiento del algoritmo. Es interesante comparar los resultados y el número de generaciones requeridas para llegar a los resultados óptimos sin y con este mecanismo.

2. MINIMIZACIÓN DE FUNCIONES

Ahora buscamos el mínimo de una serie de funciones en lugar del máximo. El conjunto de funciones que consideramos es el siguiente:

- $f(x) = 1 + \frac{cosx}{1+0.01x^2} \qquad x \in [0,50]$

  que presenta un mínimo (0.0883634) en $x = 3.08531$.

- $f(x) = -|\, xsen(\sqrt{|\, x\,|})\,| : x \in [-250,250]$

  que presenta un mínimo (-201.843) en $x = 203.814$.

- $f(x) = \frac{senx}{1+\sqrt{x}+\frac{\cos x}{1+x}} : x \in [0,\, 25]$

  que presenta un mínimo (-0.318071) en 4.5798.

- $f(x) = 5 + \exp(0.5(\pi x^2)) - \exp(0.1x\cos(\pi x)) : x \in [0,\, 25]$

  que presenta un mínimo (-5.39204) en 23.9901.

Al tratarse de una minimización es necesario realizar algunos pequeños ajustes en el algoritmo genético básico que hemos utilizado para la maximización de funciones. Ahora, tal como se ha explicado en la Sección 3.1.1 del capítulo 3,

nuestro algoritmo utiliza una adaptación revisada *f(x)* que se hace máxima cuando la función objetivo o adaptación bruta se hace mínima. Si *cmax* es el valor máximo de los valores de adaptación *g(x)* de la población en una determinada generación, entonces tenemos

$$f(x) = cmax * 1.05 - g(x)$$

También tenemos que utilizar la versión del algoritmo genético presentada en la Sección 3 que hace la revisión de la adaptación llamando a *revisar_adaptacion_minimizar* y que reproducimos aquí:

```
funcion Algoritmo_Genético_Minimiza)
{
   TPoblacion pob;     // población
   TParametros parámetros// tamaño población

   obtener_parametros(parametros);
   pob = poblacion_inicial();
   revisar_adaptacion_minimizar(pob, parámetros, cmax);
   evaluacion(pob, tam_pob, pos_mejor, sumadaptacion);

   // bucle de evolución
   mientras no se alcanza condición de terminación  hacer{
      seleccion(pob, parámetros);
      reproduccion(pob, parámetros);
      revisar_adaptacion_minimizar(pob,parámetros,cmax);
      evaluacion(pob,parámetros,pos_mejor,sumadaptacion);
   }
   devolver pob[pos_mejor]
}
```

3. OPTIMIZACIÓN DE FUNCIONES DE VARIAS VARIABLES

Buscamos ahora el óptimo de una serie de funciones que dependen de más de una variable:

- $$f(x_1, x_2) = 0.5 - \frac{-0.5 + sen(sqrt(x_1^2 + x_2^2))^2}{1.0 + 0.001(x_1^2 + x_2^2)^2}$$

  $$x_1 \in [-40,40], x_2 \in [-40,40]$$

  que presenta un máximo (1.0) en $x_1 = 0.0$, $x_2 = 0.0$.

- $$f(x_i \mid i=1..n) = 10n + \sum_{i=1}^{n} x_i^2 - 10cos(2\pi x_i) : x_i \in [-5.12, 5.12]$$

que presenta un mínimo absoluto en cero y hace que la función tome el valor cero.

- $$f(x_i \mid i=1..n) = -\sum_{i=1}^{n} sen(x_i) sen^{20}\left(\frac{(i+1)x_i^{\,2}}{\pi}\right) : x_i \in [0,\pi]$$

que presenta los siguientes mínimos en función de n :

n	mínimo
1	-1
2	-1.959091
3	-2.897553
4	-3.886358
5	-4.886358
6	-5.879585
7	-6.862457
8	-7.858851
9	-8.858851
10	-9.856182

- $$f(x_i \mid i=1..n) = \sum_{i=1}^{n} -x_i sen(\sqrt{|x_i|}) : x_i \in [0, 100]$$

que tiene un valor de $-n$ *63.63498 y se encuentra en $x_i = 65.54785$.

Cuando trabajamos con funciones de varias variables en un algoritmo genético, necesitamos una cadena de genes binaria para representar a cada una de dichas variables. Esto implica que necesitamos modificar la representación del individuo. Ahora el genotipo está formado por un vector de vectores de genes, uno por variable. El fenotipo también es un vector de reales. Necesitamos almacenar las distintas longitudes de las cadenas de genes asociadas a cada variable.

```
tipo TGenes :  vector de  booleano;
tipo Tvec_genes :  vector de TGenes;
tipo Tvec_var :  vector de real;
```

```
tipo Tvec_int :  vector de entero;
tipo TIndividuo =  registro{
  Tvec_genes vec_genes; // genotipo
  Tvec_var vec_var; // fenotipo
  Tvec_int lcrom; // longitud de la cadena
  real x; // fenotipo
  real aptitud; // función de evaluación
  real puntuacion;
 //puntuación relativa: adaptación/sumadaptación
  real punt_acu; // puntuación acumulada para sorteos
};
```

Estos cambios en la representación de los individuos conllevan también pequeñas modificaciones en el algoritmo:

- Al comienzo de la ejecución necesitamos calcular la longitud de cromosoma para cada variable.
- Al generar un individuo debemos generar el cromosoma correspondiente a cada variable.
- En la función de adaptación debemos decodificar cada una de las variables, para obtener el valor real que les corresponde y poder calcular sobre todos ellos la función de adaptación.
- El operador de cruce debe tener en cuenta todos las cadenas, cruzando x_1 del padre 1 con x_2 del padre 2, x_2 del padre 1 con x_2 del padre 2, etc.
- La mutación debe aplicarse a todas las variables.

Proyecto 2

OPTIMIZACIÓN DE FUNCIONES (B)

Este proyecto es una ampliación del proyecto anterior, y consiste en la optimización de funciones pero utilizando cromosomas con representación real. Ahora los individuos se representan mediante vectores de números en coma flotante en los que cada variable se corresponde con un gen. A continuación se muestra un ejemplo de cromosoma que representa los diferentes valores de cada una de las 5 variables que codifica en cada gen:

3.24	2.71	0.87	2.34	4.55
x_1	x_2	x_3	x_4	x_5

Algunas opciones varían ligeramente:

- **Representación de los Individuos**: se representan mediante vectores de números reales, donde cada valor real corresponde a un gen.
- **Función de evaluación**: es el resultado de evaluar la función considerada en el punto que resulta de la decodificación del individuo.
- **Selección**: por ruleta.
- **Operador de Cruce**: se propone usar los métodos de cruce de un punto, discreto uniforme, aritmético y SBX.

- **Operador de Mutación**: aleatoria de un gen dentro de los valores permitidos para ese gen.

Las funciones a estudiar son las mismas del proyecto anterior pero se han probado más a fondo las funciones de varias variables, con el fin de comparar el rendimiento con las funciones más complejas.

Como se estudió en el capítulo 4, se adapta la estructura de un cromosoma para que pueda almacenar valores reales:

```
tipo TGenes = vector de real;
tipo TIndividuo = registro{
  TGenes genes; //vector de valores reales
  real x;          // fenotipo
  real adaptación; //función de evaluación
  real puntuacion; //punt. relativa
  real punt_acu; //puntuación acumulada
  . . .
}
```

3.24	2.71	0.87	2.34	4.55
genes[0]	genes[1]	genes[2]	genes[3]	genes[4]

Debemos modificar el código para inicializar el cromosoma de forma que cada posición se inicialice con un valor de la variable dentro del rango permitido:

```
TIndividuo genera_indiv(entero lcrom) {
 . . .
 para i = 0 hasta longitudCromosoma hacer {
  genes[i]= aleaReal(limiteInf[i],limiteSup[i]);
   . . .
   . . .
 }
}
```

En los casos de funciones con una sola variable tendremos cromosomas de longitud 1. Para las funciones de varias variables utilizaremos el valor n para identificar el número de variables. Por ejemplo, si consideramos la siguiente función y un valor de $n = 10$.

$$f(x_i \mid i = 1..n) = 10n + \sum_{i=1}^{n} x_i^2 - 10cos(2\pi x_i) : x_i \in [-5.12,5.12]$$

Un posible cromosoma para el problema de minimización de la función podría ser el siguiente:

3.24	-4.22	0.23	2.71	-1.23	0.87	2.34	-4.56	-5.09	4.55
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}

Y la función de evaluación:

```
funcion eval(Tindividuo ind, entero n) {
real suma=0;
 para cada i  desde 0  hasta n  hacer{
   suma = suma+ genes[i]²-10*coseno(2*PI*genes[i]);
   }
 devolver 10*n+suma;
}
```

Y si por ejemplo consideramos la siguiente función y un valor de $n = 3$.

- $f(x_i \mid i = 1..n) = \sum_{i=1}^{n} -x_i sen(\sqrt{|x_i|}) : x_i \in [0, 100]$

que tiene un mínimo de $-n * 63.63498$ y se encuentra en $x_i = 65.54785$.

Un posible cromosoma podría ser el siguiente:

32.456	21.983	98.341
x_1	x_2	x_3

Y la función de evaluación:

```
funcion eval(Tindividuo ind, entero n) {
real suma=0;
 para cada i  desde 0  hasta n  hacer{
   suma = suma -genes[i]*seno(raizCuadrada(|genes[i]|));
   }
 devolver suma;
}
```

A continuación se muestra el pseudocódigo de algunos métodos de cruce y mutación utilizados con esta nueva representación. El cruce discreto de un punto es igual que el propuesto en el capítulo 2.

Cruce discreto uniforme

Se define una probabilidad $\mathbf{P_i}$, que permite evaluar para cada gen si los hijos heredan o no los genes intercambiados de los padres.

```
funcion cruceUniforme(TIndividuo padre1, TIndividuo padre2,
              var TIndividuo hijo1, var TIndividuo hijo2,
              entero lcrom, real probi)
 {

  . . .
  para cada i  desde 0  hasta lcrom  hacer{
     // se genera un número aleatorio en [0 1]

     prob = alea();
     si (prob < probi) entonces {
        hijo1.genes[i] = padre2.genes[i];
        hijo2.genes[i] = padre1.genes[i];
     }
     eoc
       {
        hijo1.genes[i] = padre1.genes[i];
        hijo2.genes[i] = padre2.genes[i];
       }  ...
}
```

Cruce aritmético

Implementamos el cruce aritmético en que el que el valor de cada hijo se obtiene mediante la siguiente fórmula:

$$h_i = \alpha\, p_{1i} + (1-\alpha)\, p_{2i}$$

$$0 <= \alpha <= 1$$

El valor α puede ser constante o variable para cada valor a calcular.

```
funcion cruceAritmetico(TIndividuo padre1, TIndividuo padre2,
              var TIndividuo hijo1, var TIndividuo hijo2,
              entero lcrom, real α)
{
   . . .
 para cada i desde 0  hasta lcrom  hacer{
  hijo1.genes[i]=padre1.genes[i]*α + padre2.genes[i]*(1-α);
  hijo2.genes[i]=padre2.genes[i]*α + padre1.genes[i]*(1-α);
  ...
}
```

Cruce SBX

El cruce binario simulado utiliza un número aleatorio α entre 0 y 1 que sirve para determinar el valor β.

Si α < 0.5 entonces β = $2\alpha^{(1/(n+1))}$

Si α >= 0.5 entonces β = $(1/(2(1-\alpha)))^{(1/(n+1))}$

Utilizamos un valor de n igual a 1. El valor de cada hijo se puede obtener mediante la siguiente fórmula, para cada uno de los hijos:

$$h1_i = 0.5\ (\ (P1_i+P2_i) - \beta\,|P2_i-P1_i|)$$

$$h2_i = 0.5\ (\ (P1_i+P2_i) + \beta\,|P2_i-P1_i|)$$

```
funcion cruceAritmetico(TIndividuo padre1, TIndividuo padre2,
               var TIndividuo hijo1, var TIndividuo hijo2,
               entero lcrom) {
  . . .
  alfa = alea();

  si (alfa <0.5) entonces
                    beta = 2*alfa(1/(n+1))
  eoc
       beta = 1/(2*(1-alfa)))(1/(n+1))

  para cada i desde 0 hasta lcrom hacer{

   aux1 = padre1.genes[i]+padre2.genes[i];
   aux2 = abs(padre2.genes[i]-padre1.genes[i]);
   hijo1.genes[i]=0.5*(aux1 - beta*aux2);
   hijo2.genes[i]=0.5*(aux1 + beta*aux2);
  ...
  ...

}
```

Mutación uniforme

En el caso de la mutación uniforme sobre valores reales la transformación consiste en, dado un individuo, modificar alguno de los valores cambiándolo por un valor aleatorio entre los posibles del intervalo:

```
funcion mutacion(var TPoblacion pob,  entero tam_pob,
                 entero lcrom,  real prob_mut) {
 ...
  si (prob < prob_mut){
  //generamos un valor aleatorio válido del intervalo del gen
  pob[i].genes[j] =  aleaReal(inf[i],sup[i]);
  mutado =  cierto;
  }
 ... }
```

Proyecto 3

BÚSQUEDA DE RUTAS DE METRO

1. DESCRIPCIÓN DEL PROBLEMA

En este proyecto se quiere implementar un algoritmo evolutivo para buscar el mejor camino entre dos puntos de una red de metro. La calidad del camino no sólo depende del número de estaciones y trasbordos que contiene, sino también de cumplir determinadas restricciones sobre estaciones por las que se desea pasar o que se quieren evitar. A veces se encuentran soluciones válidas, aunque no óptimas, y a veces ni siquiera existe una solución que respete todas las restricciones.

Los algoritmos tradicionales no aseguran una solución óptima y rápida para la búsqueda de caminos en grafos. Por esta razón la programación evolutiva es una alternativa a considerar.

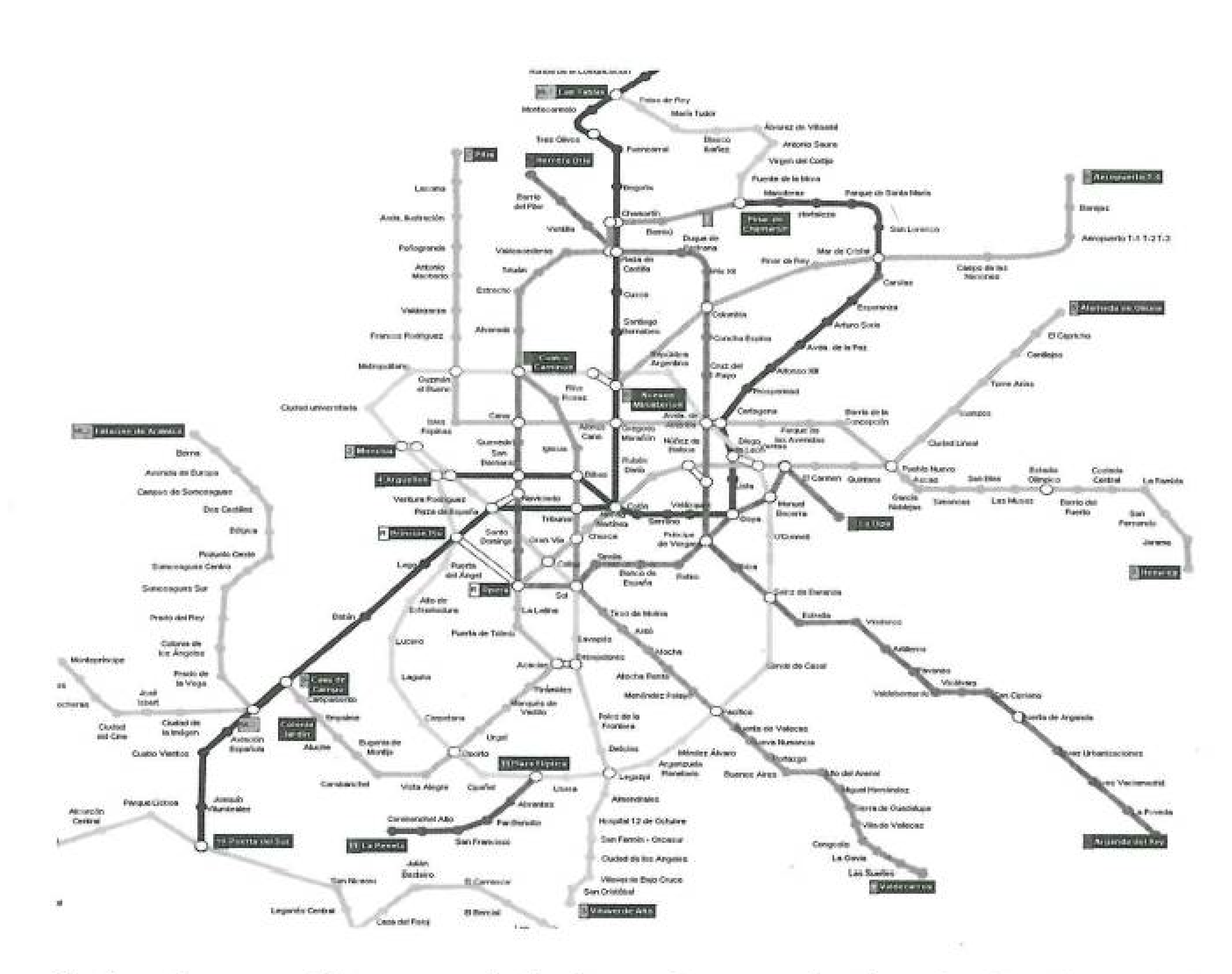
El algoritmo debe tener en cuenta las siguientes restricciones:

- El trayecto debe empezar y terminar en las estaciones especificadas, de forma que recorra el menor número posible de estaciones, y realizando el menor número posible de trasbordos.

- El trayecto debe evitar pasar por las estaciones que se indique, y debe pasar por otras estaciones indicadas. Estas restricciones pueden considerarse más o menos prioritarias. El algoritmo debe permitir especificar su importancia.

Entradas al programa

- Plano de metro:



Se lee de un archivo con el siguiente formato: la descripción de cada línea de metro comienza con una línea del archivo con el nombre de la línea. A continuación hay una línea de archivo para cada estación de la línea, que comienza con el nombre de la estación, seguida de la lista de nombres de líneas con las que tiene correspondencia y terminada en un *. Los datos de una línea terminan con una línea de archivo con la marca #. Por ejemplo:

```
Linea1 N

Congosto *
Villa_de_Vallecas *
Sierra_de_Guadalupe *
Miguel_Hernandez *
Alto_del_Arenal *
Plaza_de_Castilla Linea9 Linea10 *
...
```

```
#
Linea2 N
Cuatro_Caminos Linea1 Linea6 *
Canal Linea7 *
Quevedo *
San_Bernardo Linea4 *
...
```

- Datos de la consulta:

Los datos del trayecto que se quiere realizar pueden introducirse interactivamente o leerse de un archivo. Hay que especificar las estaciones inicial y final del trayecto, así como las estaciones por las que se desea pasar y las estaciones por las que no debe pasar.

La salida es una secuencia de estaciones, especificando en cada una a qué línea de metro pertenece. Los resultados, además de salvarse en un archivo, deben presentarse de forma que el usuario, de forma cómoda, pueda saber la longitud del trayecto, el número de trasbordos a realizar y si el trayecto respeta o no las restricciones solicitadas.

Observaciones

- Un trayecto no debe pasar dos veces por la misma estación.
- Hay que tener en cuenta que pueden existir líneas circulares.
- Hay que tener en cuenta que pueden existir correspondencias entre estaciones con distinto nombre. Por ejemplo, en la red de metro de Madrid, hay una correspondencia entre "Plaza de España" y "Noviciado".

2. DISEÑO DEL ALGORITMO

En primer lugar necesitamos definir la forma de representar la red de metro en la que se van a buscar los caminos. Una posibilidad es representar el plano de metro como un conjunto o lista de líneas de metro:

```
// Representación del PLANO de metro completo
tipo TPlano : vector de TLinea;
```

A su vez, cada línea podemos representarla como una secuencia o lista de estaciones. Además, asignamos un nombre a cada línea y un indicativo de si es o no circular.

```
tipo TLEstaciones: vector de TEstacion;

tipo TLinea = registro{
   cadena_caracteres nombre; // nombre asignado a la línea
   TLEstaciones estaciones; // secuencia de estaciones
                            // la línea
   booleano circular; // indicativo de línea circular o no
}
```

Para representar las estaciones utilizamos una estructura de datos que incluya información sobre el nombre de la estación, sobre el posible nombre que puede tener esa estación en otra línea y sobre las correspondencias con otras líneas a las que se puede cambiar en esa estación.

```
tipo TEstacion = registro{
   //nombre de la estación en esa línea
   cadena_caracteres nombre;
   //nombre de la estación en otra línea
   cadena_caracteres alias;
   //líneas con las que se cruza
   TCorrespondencias corresp;
}
```

Las correspondencias de una estación las podemos representar simplemente como un vector de nombres o cadenas de caracteres.

```
tipo TCorrespondencias : vector de cadena_caracteres;
```

También necesitamos representar los datos de una consulta que se realiza al algoritmo. Una consulta incluye la estación inicial del trayecto, la final, un conjunto de nombres de estaciones por las que no se quiere pasar y un conjunto de nombres de estaciones por las que se desea pasar.

```
tipo TLNombres : vector de cadena_caracteres;
 tipo TConsulta = registro{
 // nombre de la estación inicial
 cadena_caracteres est_inicial;
```

```
  // nombre de la estación final
  cadena_caracteres est_final;
  TLNombres est_prohibidas; // estaciones a evitar
  TLNombres est_obligadas; // estaciones por las que pasar
}
```

2.1 Representación de los individuos

Sabemos que la solución a nuestro problema será una secuencia de estaciones del plano de metro considerado, que comience en la estación inicial del trayecto buscado y termine en la estación final de dicho trayecto. Por eso, cada individuo representa un trayecto entre la estación inicial y la final. Como hay estaciones que pertenecen a distintas líneas, cada estación especificada en un individuo debe incluir información sobre la línea a la que pertenece.

Por lo tanto, podemos representar a los individuos como listas de genes en las que cada gen representa a una estación del recorrido y a la línea a la que corresponde.

```
tipo Tgen = registro{
  // posición en la lista de líneas del plano
  entero linea;
  // posición de la estación en la línea
  entero estación;
}
```

Luego, cada individuo se representa como una cadena de genes como los descritos, junto con la información usual que necesita un algoritmo evolutivo.

```
tipo TGenes : vector de TGen;
tipo TIndividuo = registro{
  TGenes genes; // cadena de genes(genotipo)
  real adaptación; // función de evaluación
  real puntuacion; // puntuación relativa
  real punt_acu; // puntuación acumulada para sorteos
}
```

2.2 Generación de la población inicial

Para reducir el espacio de búsqueda y hacer más eficiente el algoritmo, es conveniente que las secuencias de estaciones de los individuos de la población inicial, además de empezar y terminar en las estaciones inicial y final del trayecto

buscado, cumplan otras condiciones. Una de estas condiciones es que los trayectos no pasen dos veces por la misma estación, porque esto dará lugar a tramos de recorrido inútiles.

También para reducir el espacio de búsqueda podemos establecer un límite a la longitud de un individuo, pasado el cual si no se ha llegado a la estación final se empieza a generar de nuevo.

La idea de la generación de un nuevo individuo es partir de la estación inicial del recorrido, e ir generando estaciones por las que puede continuar, hasta llegar a la estación final, y teniendo en cuenta las condiciones mencionadas. De acuerdo con estas consideraciones, podemos seguir el siguiente esquema para generar los individuos de la población inicial:

- Se buscan las líneas en las que se encuentra la estación inicial y se selecciona una de ellas aleatoriamente.
- Para evitar que los trayectos avancen y retrocedan por una misma línea, es necesario establecer una dirección de movimiento (por ejemplo con una variable booleana *avanza* que toma el valor cierto para indicar avance o falso para indicar retroceso).
    - Si la estación inicial está al comienzo o al final de la línea sólo hay una posible dirección, que queda así automáticamente determinada.
    - Si la estación inicial es una estación intermedia, se elige una dirección aleatoriamente.
- De acuerdo con la dirección de avance del movimiento, se genera una secuencia de estaciones contiguas, hasta llegar a la estación final o al límite establecido para el trayecto. El procedimiento en cada paso o estación es el siguiente:
    - Se pasa a la siguiente estación de la línea de acuerdo con la dirección establecida del movimiento. Hay que tener en cuenta que si se trata de una línea circular, al llegar a la última estación de la lista que la representa se pasa a la primera, y al llegar a la primera en un movimiento de retroceso, se pasa a la última.
    - Se comprueba que la nueva estación no estuviera ya incluida en el trayecto del individuo. Si lo está, y la otra dirección de movimiento es posible, se intenta. Si no se deshecha el individuo.

- Para la nueva estación:
    - Si hay correspondencias se decide aleatoriamente si se continúa en la misma línea o si se cambia.
    - Si se cambia se elige aleatoriamente una de las correspondencias, y en la nueva línea se elige una dirección.

2.3 Función de adaptación

El objetivo es minimizar el número de estaciones del recorrido, pero teniendo en cuenta otras restricciones. Por ejemplo, podemos incluir una penalización por cada trasbordo realizado. Así mismo se penalizará la violación de las restricciones de inclusión o exclusión de las estaciones especificadas. Las penalizaciones deben ser parámetros de entrada al algoritmo.

En función de los valores que se les asigne, primaremos un tipo u otro de trayecto: con pocos transbordos, que nunca pasen por una de las estaciones que se desea evitar, o que pueda pasar si así se reduce considerablemente la longitud del trayecto, etc. La siguiente tabla muestra un ejemplo de posibles valores:

```
constante entero PENAL_TRANSBORDO 1
constante entero PENAL_PROHIB 10
constante entero PENAL_OBLIG 3
```

Un posible esquema de la función de adaptación será el siguiente:

```
funcion adaptacion(TIndividuo individuo, TPlano plano,
                   TConsulta consulta, ...
{
   real adapt;
   entero num_transb; // número de transbordos
   entero num_prohib; // número de estaciones a evitar
   entero num_obliga; // número de estaciones a visitar

   // se inicializa la adaptación a la longitud
   // del trayecto
   adapt = tamaño(individuo.genes);
   // se cuenta el número de transbordas del trayecto
   num_transb = { ··· }
   adapt = adapt + num_transb * PENAL_TRANSBORDO;

   // se cuenta el número de estaciones prohibidas
   // por las que pasa el trayecto
```

```
    num_prohib = { ··· }
    adapt = adapt + num_prohib * PENAL_PROHIB;

    // se cuenta el número de estaciones por las
    // que debe pasar y no pasa

    num_oblig = { ··· }
    adapt = adapt + num_oblig * PENAL_OBLIG;

    devolver adapt;
}
```

2.4 Operador de cruce

Los operadores genéticos deben tener en cuenta que deben generar individuos válidos. Por ejemplo, el operador de cruce puede buscar una estación en común entre los individuos a cruzar e intercambiar los segmentos a ambos lados de dicha estación. Un posible procedimiento para buscar un punto de cruce válido en los dos individuos es el siguiente:

- Se toman dos individuos a cruzar, con el procedimiento descrito en el capítulo 2.
- Se elige al azar una posición, el punto de cruce1, de la secuencia de estaciones que define el trayecto del primero de los individuos.
- Se busca en el otro individuo la estación correspondiente. Si se encuentra su posición es el punto de cruce2.
- Si no se encuentra, el punto de cruce1 avanza a la siguiente posición, y se repite el proceso. El avance del punto de cruce1 se realiza de forma circular, hasta volver al punto de partida. Es decir, si se llega al final de la secuencia de estaciones se pasa a la primera.
- Si después de comprobar todas las estaciones del primero de los individuos no se ha encontrado ninguna estación común que permita realizar el cruce, no se aplica el operador.

Hay que tener en cuenta que la posibilidad de que el cruce no se llegue a aplicar, si los dos individuos seleccionados no comparten ninguna estación, obliga a utilizar una tasa de cruce mayor que las que se utilizan en otros algoritmos evolutivos.

2.5 Operador de mutación

El operador de mutación selecciona aleatoriamente una estación del trayecto y genera un nuevo trayecto desde ella hasta la estación final o bien de la estación inicial a la seleccionada. Es decir, la aplicación de este operador sigue los siguientes pasos:

- Se elige al azar una posición, el punto de mutación, de la secuencia de estaciones que define el trayecto del individuo.
- Se decide aleatoriamente si se renueva el trayecto entre la estación inicial y el punto de mutación o entre el punto de mutación y la estación final.
- Se genera un nuevo trayecto para el tramo elegido, de la misma forma que se han generado los trayectos de la población inicial, pero entre las estaciones que corresponda en este caso.

2.6 Consideraciones adicionales

Como es habitual en los algoritmos evolutivos, es necesario hacer un estudio de los parámetros del algoritmo:

- Tamaño de la población
- Número límite de iteraciones del algoritmo
- Porcentaje de cruces
- Porcentaje de mutaciones

para alcanzar un rendimiento óptimo del sistema. En este caso, además de los parámetros también es interesante analizar y obtener resultados sistemáticos, de los valores de las penalizaciones que nos permiten obtener determinados tipos de trayectos.

Por ejemplo, podemos buscar los valores que nos permitan:

- Garantizar que se minimiza el número de transbordos realizados, quizás a costa de violar alguna de las restricciones de pasar o dejar de pasar por las estaciones prohibidas u obligadas.

- Garantizar que no se pasa por ninguna de las estaciones prohibidas, quizás a costa de obtener trayectos más largos o con más transbordos.

- Garantizar que no se pasa por ninguna de las estaciones prohibidas, ni se deja de pasar por ninguna de las obligadas.

3. TRATAMIENTO ALTERNATIVO DE LAS RESTRICCIONES

El proyecto abordado en este capítulo involucra diversas restricciones. Algunas de ellas, como el requisito de que los trayectos empiecen y terminen en las estaciones especificadas en la consulta del usuario, las hemos introducido en la codificación del individuo. No permitimos que existan individuos que no cumplan esta condición.

El resto de las restricciones, que hemos considerado más flexibles, como minimizar el número de transbordos, procurar que el trayecto pase por determinadas estaciones o deje de pasar por otras, se han tratado con técnicas de penalización. Sin embargo, existen otras alternativas.

Puede resultar interesante estudiar cómo afecta a los resultados y a la eficiencia del sistema las siguientes alternativas:

- Permitir que algunos (si se hace con todos el algoritmo sería lentísimo) individuos empiecen y terminen en estaciones aleatorias, penalizando a los que no correspondan a las estaciones del trayecto, y de manera que la penalización se vaya incrementando a medida que avanza la evolución.

- Introducir en la codificación las restricciones de estaciones prohibidas. Al generar los trayectos evitar las prohibidas hasta cierto número de intentos.

4. CON OTRAS RESTRICCIONES

También podemos refinar el proyecto introduciendo especificaciones más detalladas de la red de metro. Algunas posibilidades son:

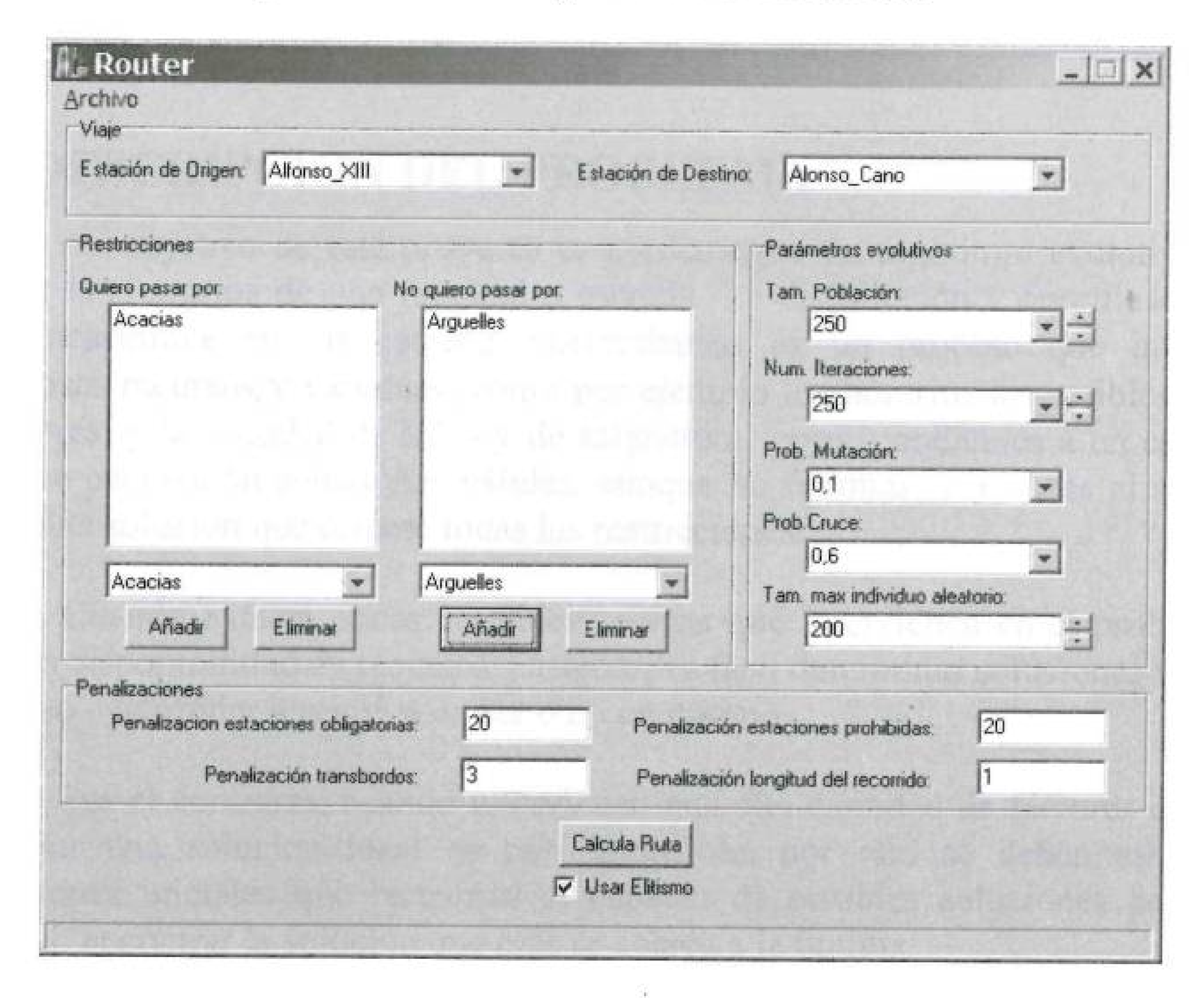
- Asignar un coste específico a cada transbordo: la distancia a recorrer por el usuario puede ser muy variable de unos transbordos a otros.

- Especificar la distancia entre estaciones.
- Especificar la velocidad media de los trenes en cada una de las líneas, que puede variar de unas a otras.

5. UN EJEMPLO DE INTERFAZ GRÁFICA

A continuación se muestra un ejemplo de aplicación que incluye una interfaz gráfica que permite seleccionar diferentes parámetros para el algoritmo, así como visualizar la solución obtenida.

La aplicación permite seleccionar los parámetros del algoritmo: estaciones de origen y destino, restricciones, penalizaciones, tamaño de la población, número máximo de generaciones, probabilidad de cruce y mutación, porcentaje de elitismo, método de selección, método de cruce y método de mutación.



Al terminar la ejecución la aplicación muestra la ruta obtenida:

Ruta

Linea Origen	Estacion Origen	Línea Dest...	Estacion Destino	Accion
Linea4	Alfonso_XIII	Linea4	Prosperidad	
Linea4	Prosperidad	Linea4	Avda_de_America	Trasbordo a: Linea7
Linea7	Avda_de_America	Linea7	Gregorio_Marañon	
Linea7	Gregorio_Marañon	Linea7	Alonso_Cano	

Proyecto 4

PLANIFICACIÓN DE HORARIOS

1. DESCRIPCIÓN DEL PROBLEMA

El objetivo de este proyecto es implementar un algoritmo evolutivo para diseñar los horarios de una facultad o escuela. La distribución y coordinación de carga académica en las carreras universitarias es un proceso que involucra numerosos recursos y variables, como por ejemplo los horarios disponibles de los profesores, y la cantidad de aulas y de asignaturas correspondientes a un curso. A veces se encuentran soluciones válidas, aunque no óptimas, y a veces ni siquiera existe una solución que respete todas las restricciones.

Cuando existen pocas variables críticas que intervienen en el problema y una alta disponibilidad de recursos a asignar, es fácil determinar soluciones válidas, e incluso determinar a priori si existe o no un óptimo.

Por el contrario, cuando intervienen una alta cantidad de factores críticos, encontrar una solución ideal es casi imposible, por ello se deben establecer condiciones iniciales que restrinjan el espacio de posibles soluciones pero que permitan encontrar la solución que más se acerca a la óptima.

Los algoritmos tradicionales no aseguran una solución óptima y rápida para la asignación de horarios. Por esta razón la programación evolutiva es una alternativa a considerar.

El algoritmo debe tener en cuenta las siguientes restricciones:

- Dos clases no pueden darse simultáneamente en la misma aula.
- Un profesor no puede impartir clase simultáneamente en dos aulas.
- Cada profesor tiene unos horarios disponibles, y entre estos establece preferencias.

Entradas al programa

- Lista de asignaturas a las que hay que asignar horario:

Código asignatura	Nombre asignatura
101	S1
102	S2
...	...

- Lista de profesores con las asignaturas que imparte cada uno y sus disponibilidades y preferencias de horario.

Por ejemplo, a cada horario se le asigna un coste cuyo valor es 0 si es un horario posible y preferente, y 1 si es posible pero inconveniente.

Cód prof.	Nombre prof.	Cód. asig	Horario	Coste
20	P1	101	H1	0
20	P1	101	H3	1
20	P1	101	H4	0
20	P1	101	H2	1
20	P1	105	H2	0
...	...	...	...	...

- Lista de aulas disponibles para cada uno de los horarios.

Nombre aula	Horario
A1	H1
A1	H2
A1	H3
A2	H2
...	...

Estas entradas se leen de archivos. La salida es una tabla, con una asignación de horario para cada una de las asignaturas. Los resultados, además de salvarse en un archivo, pueden presentarse en pantalla, de forma que el usuario, de forma interactiva, puede consultar el horario asignado a un profesor en una o en todas sus asignaturas, las aulas asignadas a una asignatura, etc.

2. DISEÑO DEL ALGORITMO

Se trata de un problema de minimización, para el que aplicaremos el esquema de algoritmo evolutivo que corresponde. Es también un problema de restricciones que trataremos con la técnica de penalización.

Comencemos por considerar una posible representación de los datos de entrada al problema. Tenemos que representar información sobre aulas, profesores y asignaturas, procurando facilitar el acceso desde cada uno de estos datos a aquellos con los que se relaciona. Comencemos por los profesores. Para cada uno de ellos tenemos que representar un código asignado, su nombre y la lista de asignaturas que se encarga de impartir, que es un dato que no puede variarse. Las asignaturas de un profesor se indican por su código, y para cada una de ellas se tiene una lista de posibles horarios y un coste asignado a cada horario, según las preferencias del profesor.

Podemos representar los costes de los horarios de un profesor para una determinada asignatura mediante la estructura de datos (o clase) *TLCost_Hora_Prof*, que consiste en una lista de posibles horarios para que el profesor imparta la asignatura considerada y su coste asociado.

```
tipo TLCost_Hora_Prof: vector de TCost_Hora_Prof;
tipo TCost_Hora_Prof = registro{
  cadena_caracteres horario; // nombre del horario
  entero coste; // coste del horario
}
```

Representamos ahora una lista de asignaturas, identificadas por sus códigos, a cada una de las que se asocia su lista del tipo *TLCost_Hora_Prof*. Se trata del tipo *TLCost_Asig_Prof*, que es una lista de estructuras *TCost_Asig_Prof*:

```
tipo TCost_Asig_Prof = registro{
   entero codigo_asig; // código de la asignatura
   TLCost_Hora_Prof Horarios_Coste; // lista de posibles
   // horarios con sus costes para la asignatura
}
tipo TLCost_Asig_Prof: vector de TCost_Asig_Prof;
```

Finalmente, podemos representar a un profesor con la estructura *TProfesor*, incluyendo su nombre, su código y su lista de asignaturas asignadas con sus posibles horarios y costes asociados. *TProfesores* representa la lista de todos los profesores.

```
tipo TProfesor = registro{
   cadena_caracteres nombre; // nombre del profesor
   entero codigo; // código del profesor
   TLCost_Asig_Prof conj_asig;
   // asignaturas (con su coste por horario)
   // asignadas al profesor
}

tipo TProfesores: vector de TProfesor;
// lista de profesores
```

La representación de cada aula incluye su nombre y la lista de nombres de horarios en los que está disponible. El conjunto de todas las aulas se representa por la estructura *TLAulas*, que es una lista de estructuras *TAula*.

```
tipo TLHorarios: vector de cadena_caracteres; // lista
de nombres de horarios
// Estructura para representar
// la información sobre una aula

tipo TAula = registro{
   cadena_caracteres nombre; // nombre del aula
   TLHorarios horarios;
// horarios en los que está disponible
}

tipo TLAulas: vector de TAula; // lista de aulas
```

Por su parte la representación de las asignaturas incluye el nombre, el código de la asignatura y el profesor que la imparte. Para hacer más eficientes los accesos a la información sobre el profesor, éste se representa por su posición en la lista de profesores. La estructura para representar la información sobre una asignatura es la siguiente:

```
tipo TAsignatura = registro{
   cadena_caracteres nombre; // nombre de la asignatura
   entero codigo; // código de la asignatura
   entero pos_prof;

// posición (en la lista de profesores)
// del profesor asignado a la asignatura

}
tipo TLAsignaturas: vector de TAsignatura;
// lista de asignaturas
```

2.1 Representación de los individuos

Los individuos pueden representar una asignación de horarios a cada una de las asignaturas a considerar. Por lo tanto, podemos representar a los individuos como listas de genes en las que cada gen representa a una asignatura y el horario asignado a ella. Incluimos también el coste asociado para hacer más eficiente el cálculo de la adaptación:

```
tipo Tgen = registro{
   entero cod_asig; // código de la asignatura
   entero coste; // coste asociado
   cadena_caracteres horario; // nombre del horario
}
```

Luego, cada individuo se representa como una cadena de genes como los descritos, junto con la información usual que necesita un algoritmo evolutivo.

```
tipo TGenes : vector de  TGen;
tipo TIndividuo = registro{
   TGenes genes; // cadena de genes(genotipo)
   real adaptación; // función de evaluación
   real puntuacion;//punt. rel.:adaptación/sumadaptación
   real punt_acu; // puntuación acumulada para sorteos
}
```

2.2 Generación de la población inicial

Los individuos de la población inicial se generan asignando a cada asignatura un horario aleatorio de entre los posibles que tiene disponibles el profesor de la asignatura que la imparte. Un posible esquema es el siguiente:

- Para cada asignatura:
  - Se busca el profesor asignado para impartirla.
  - Se busca la asignatura entre las del profesor.
  - Se elige aleatoriamente uno de los posibles horarios de esa asignatura entre los horarios del profesor.
  - Se asigna el coste que le corresponde.

2.3 Función de adaptación

La función de evaluación debe penalizar cada violación de las restricciones del problema y considerar el coste que supone para cada profesor cada uno de sus horarios disponibles.

Un posible esquema de la función de adaptación es el siguiente:

```
funcion adaptacion(TIndividuo individuo, ...
{
   real adapt;
   entero nph, num_hor;
         ...

   adapt = 0;
   // se suma el coste de los horarios asignados
   // a cada asignatura
  para cada asignatura del individuo hacer
     adapt = adapt + individuo.genes[i].coste;

   // se penalizan las situaciones IMPOSIBLES
   // un profesor no puede estar en dos aulas a la vez

  para cada asignatura del individuo hacer{
     prof = profesor de la asignatura
     nph = num. de apariciones de prof en el mismo horario;
     si nph > 1 entonces
        adapt = adapt + (nph - 1)* PENALIZACION;
  }
```

```
   // no puede haber dos asignaturas en la
   // misma aula y horario
   para cada asignatura del individuo hacer{
      hor = horario de la asignatura
      num_hor = num. de apariciones de hor;
      si num_hor > num_aulas entonces
         adapt = adapt + (num_hor - num_aulas)*
      PENALIZACION;
   }
   devolver adapt;
}
```

La adaptación de un individuo, que en este proyecto es un valor a minimizar, recoge la suma de los costes asociados a los horarios asignados a cada asignatura, que se corresponden con las preferencias de los profesores. A este valor se le añade una penalización, dada por el parámetro *PENALIZACION*, por cada situación imposible a que da lugar la asignación de horarios del individuo. En primer lugar se comprueba la existencia de situaciones en que un profesor está a la vez en dos aulas. La variable *nph* lleva cuenta del número de veces que aparece un profesor en el mismo horario. Cada vez que *nph* es mayor que 1, se penaliza la adaptación. Después se comprueba que no haya dos asignaturas en la misma aula y horario. Para hacerlo se cuenta el número de veces que se usa un mismo horario, y si este valor es mayor que el número de aulas, se penaliza la adaptación.

2.4 Operador de cruce

Podemos aplicar un sencillo cruce monopunto que intercambie en los hijos los horarios asignados en los padres antes y después de la asignatura elegida como punto de cruce.

También podemos aplicar un cruce multipunto e intercambiar el segmento de horarios que se encuentre entre dos puntos de cruce elegidos aleatoriamente.

2.5 Operador de mutación

Un sencillo operador de mutación que podemos utilizar consiste en seleccionar aleatoriamente una posición o gen del individuo y cambiar el horario asignado a la asignatura de esa posición. El nuevo horario también debe estar entre los posibles para el profesor que imparte la asignatura.

2.6 Consideraciones adicionales

Para realizar un estudio de los parámetros más adecuados para el algoritmo se puede implementar un sistema que permita variar los parámetros (tamaño de la población, número límite de iteraciones del algoritmo, porcentaje de cruces, porcentaje de mutaciones) interactivamente.

En este proyecto también es interesante variar la penalización asociada a la violación de restricciones. Por ejemplo hay que estudiar qué penalización es necesaria aplicar al caso en que se asigna a un profesor un mismo horario para dos asignaturas para garantizar que no se dé el caso, salvo si el problema no tiene solución.

Hay que tener en cuenta que usar una penalización excesiva para las violaciones de restricciones también puede dar lugar a malos resultados. Si el valor de esta penalización es órdenes de magnitud superior a los costes asociados a las preferencias de los profesores, el mecanismo de selección no tendrá en cuenta dichas preferencias, ya que las diferencias entre unos y otros individuos con una penalización por violación de restricciones serán despreciables.

3. TRATAMIENTO ALTERNATIVO DE LAS RESTRICCIONES

En lugar de considerar las restricciones del problema con técnicas de penalización, pueden probarse otros enfoques, y comparar los resultados obtenidos en ambos casos.

Podemos intentar generar individuos que no violen las restricciones del problema. Por ejemplo, podemos considerar un esquema de generación de individuos de la población inicial como el siguiente:

- Para cada asignatura:
    - Se busca el profesor asignado para impartirla.
    - Se busca la signatura entre las del profesor.
    - Mientras no se asigne horario y quede alguno por probar hacer lo siguiente:

    - Se elige aleatoriamente uno de los posibles horarios de esa asignatura y que no se haya probado antes, y se marca como candidato.

    - Se comprueba que en ese horario el profesor no tenga otra asignatura. Si es así se pasa a probar otro horario.

    - Se comprueba que en ese horario no estén todas las aulas ocupadas. Si es así se pasa a probar otro horario.

- Si se ha conseguido asignar horario se asigna el coste que le corresponde. En otro caso se deshecha el individuo.

Este método tendrá dificultades para encontrar soluciones en problemas muy restringidos.

4. CON OTRAS RESTRICCIONES

Podemos considerar versiones más sofisticadas del problema, que tengan en cuenta otros elementos que también son importantes en la planificación de horarios. Algunas posibilidades son:

- Tener en cuenta el tamaño de las aulas, y el tamaño de los grupos de alumnos de cada asignatura.

- Permitir especificar otras preferencias, aparte de las de los profesores. Por ejemplo, preferencia por que una asignatura que se imparte varios días a la semana lo haga siempre en el mismo horario.

- Control del número máximo de horas diarias de clase para los alumnos y quizás también para los profesores.

Proyecto 5

CORTADO DE PATRONES

1. DESCRIPCIÓN DEL PROBLEMA

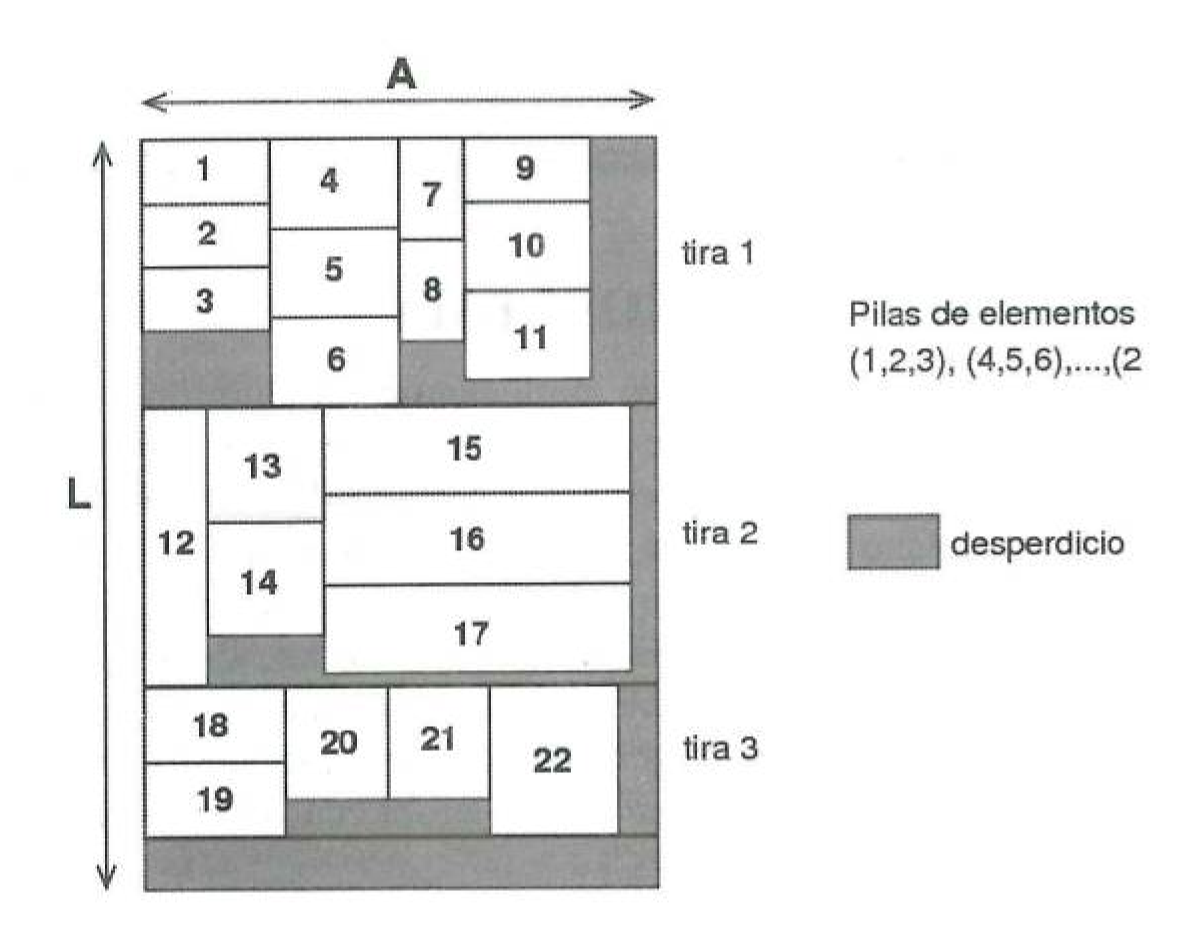
El objetivo de este proyecto es implementar un algoritmo evolutivo para planificar el cortado de un conjunto de patrones maximizando el aprovechamiento del material. Concretamente, el problema del cortado de patrones bidimensional consiste en cortar un plano rectangular en elementos rectangulares más pequeños, de los tamaños y valores especificados, de manera que se minimice el material desperdiciado. Este tipo de problema aparece en el cortado de planchas de acero, de vidrio y de madera en las industrias correspondientes. Concretamente, aquí desarrollaremos una adaptación del algoritmo evolutivo propuesto por Puchinger et al. [PUC04] en un artículo de investigación. El problema se define de la siguiente forma:

- Un conjunto de hojas rectangulares idénticas de anchura $W>0$ y largo $L>0$.

- Un conjunto de tipos de elementos rectangulares $\varepsilon = \{E_1,..,E_m\}$, que solicita el cliente. Cada tipo de elemento E_i, $i = 1...m$, se define por la anchura de los elementos, w_i, $(0<w_i \leq W)$, su longitud, l_i, $(0<l_i \leq L)$, y el número $n_i > 0$ de elementos de este tipo que se solicitan.

- El objetivo es encontrar patrones de cortado de todos los elementos solicitados de cada tipo utilizando el menor número posible de hojas, es decir, minimizando el desperdicio.

Esta versión general del problema da lugar a numerosos casos especiales en función de las restricciones adicionales que se presenten. Uno de estos casos se da cuando los cortes se realizan mediante una *guillotina*, es decir, que van de un extremo al opuesto del rectángulo. Esta restricción aparece frecuentemente en la práctica, especialmente en el cortado de planchas de papel y de vidrio. Otra restricción habitual es dividir el proceso en distintas etapas, para poder refinar el tamaño de los cortes de guillotina. Nosotros consideramos un proceso de producción de tres etapas. La primera etapa sólo corta hojas enteras horizontalmente en una serie arbitraria de *tiras*. En la segunda etapa se procesan dichas tiras cortándolas verticalmente en un número arbitrario de *pilas*. La tercera etapa realiza de nuevo cortes horizontales, produciendo los elementos finales (y el desperdicio) a partir de las pilas. La figura muestra un ejemplo de un rectángulo cortado en tres etapas mediante cortes de guillotina.



Cualquier patrón válido de cortado en tres etapas se puede reducir siempre a su denominada *forma normal* moviendo cada elemento a la posición más a la izquierda y superior posible. De esta forma, el desperdicio sólo aparece en la parte inferior de las pilas, a la derecha de la última pila de una tira o en la parte que queda por debajo de la última tira de cada hoja. El patrón que aparece en la figura está en forma normal. Nosotros consideramos sólo patrones de corte en forma normal, lo que reduce el espacio de búsqueda general infinito de patrones arbitrarios en tres etapas a un número finito de soluciones posibles. Para un patrón

de cortado dado, los elementos se producen siempre procesando de arriba a abajo las tiras dentro de cada hoja, las pilas dentro de cada tira de izquierda a derecha, y los elementos dentro de cada pila de arriba a abajo. La numeración de los elementos de la figura denota este orden.

El formato de los ficheros con los datos de prueba es el siguiente:

ANCHURA ALTURA (de las hojas de material)

ANCHURA ALTURA CANTIDAD (de un elemento requerido)

ANCHURA ALTURA CANTIDAD (de un elemento requerido)

...

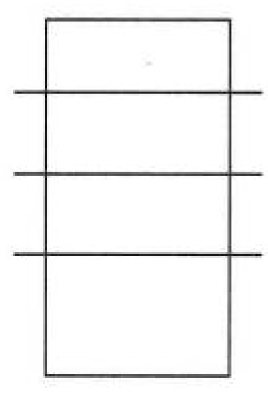
donde la primera fila indica las características de las hojas de material, y las siguientes filas están dedicadas a cada uno de los tipos de elementos requeridos. Cada una de ellas especifica la altura y anchura del tipo de elemento y la cantidad de elementos del tipo que se solicitan.

2. DISEÑO DEL ALGORITMO

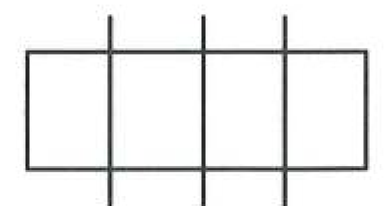
El objetivo es producir una colección de objetos rectangulares, cuyas dimensiones particulares son una entrada al problema, cortándolos de planchas de material de la anchura y longitud especificadas también en la entrada al problema, de forma que se desperdicie la menor cantidad posible de material.

El **mecanismo de cortado** es el correspondiente a una guillotina, lo que implica que todos los cortes van de un extremo a otro del rectángulo cortado. Consideramos tres etapas en el cortado:

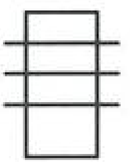
- La **hoja** de material se corta en **tiras**.



- Cada **tira** se corta en **pilas**.



- Cada pila se corta en elementos.



2.1 Representación de los datos de entrada

En primer lugar necesitamos representar los datos de entrada al problema: la hoja de material con sus dimensiones y la lista de tipos de elementos.

```
tipo TForma = registro{ // hojas de material
   real ancho;
   real alto;
}

tipo TTipo_elem = registro{
   real ancho;
   real alto;
   entero cantidad;
}
```

Cada elemento consta de los mismos datos que una hoja de material, es decir, ancho y alto. Por claridad utilizaremos un tipo específico para representarlo:

```
tipo TElem = TForm;

// Lista de los tipos de elementos requeridos
tipo TLTipos: vector de Ttipo_elem;
```

2.2 Representación de los individuos

El siguiente paso en la implementación del algoritmo es la representación de los individuos de la población. Cada individuo debe representar un patrón de cortado válido que contenga todos los elementos requeridos. Una posible representación es utilizar un vector de enteros que indique el orden en el que se colocan los elementos en las hojas, es decir, cada gen es una referencia (posición) a la lista de elementos. De acuerdo con esto, la representación que adoptamos es un vector de **enteros**, no de booleanos, que no sería una representación natural para el problema.

```
tipo TGenes: vector de integer;
```

El fenotipo, o patrón de cortado, correspondiente se obtiene aplicando una estrategia avara de asignación de elementos, que describimos en el apartado dedicado a la función de adaptación. Podemos mejorar la eficiencia del algoritmo si almacenamos el fenotipo correspondiente al genotipo también en el individuo. En nuestro caso el fenotipo es el patrón de cortado que se deriva del orden especificado en el genotipo. La construcción del fenotipo es necesaria para calcular la función de adaptación. Puesto que ésta es la parte del algoritmo más costosa computacionalmente, podemos aprovechar para dejarla almacenada si la vamos a necesitar para otros propósitos, como es la representación visual del patrón de cortado, durante la depuración del algoritmo, o para la presentación de la solución final. El tipo de dato utilizado para representar el patrón de cortado se describe también en el apartado dedicado al cálculo de la adaptación de un individuo.

```
tipo TIndividuo = registro{
   TGenes genes; // cadena de enteros (genotipo)
   real aptitud; // función de evaluación
   real puntuacion;
   //puntuación relativa: adaptación/sumadaptación

   real punt_acu; // puntuación acumulada para sorteos
   TCortado cortado; // fenotipo
 };
```

La población es una colección de individuos:

```
tipo TPoblacion: vector de TIndividuo;
```

Una representación alternativa, que es la que adoptamos aquí, es utilizar un vector de enteros que indique el orden en el que se colocan los elementos de cada tipo en las hojas, suponiendo que todos los elementos del mismo tipo se colocan sucesivamente de acuerdo al orden impuesto por la forma normal. Es decir,

asignamos un gen no a cada elemento sino a cada tipo. El valor de este gen es la posición del tipo en la lista de tipos.

Esta representación es ventajosa cuando hay varios elementos del mismo tipo, lo que suele ocurrir en las especificaciones de un cortado real. Colocando agrupados los elementos de un mismo tipo suele ahorrarse espacio. Desde el punto de vista de la implementación las diferencias de adoptar una u otra representación son pequeñas. Al ir a calcular la función de adaptación hay que tener en cuenta si los elementos están especificados en una lista de tipos. En la representación por elementos conviene añadir operadores genéticos adicionales que tiendan a agrupar elementos similares, para mejorar la calidad de los individuos.

Una práctica interesante es realizar un estudio comparativo de los resultados obtenidos con ambos tipos de representaciones.

2.3 Generación de la población inicial

Los individuos son permutaciones de la lista de tipos (o en su caso de la de elementos). Para construirlos seguimos el siguiente proceso:

- Se forma un vector con el identificativo de cada elemento (o tipo).

- Se generan números aleatorios (entre 1 y la longitud del vector) para indicar la posición del vector de la que se toma el siguiente elemento que pasa al individuo.

Por ejemplo, supongamos que tenemos una longitud de cadena de 5. Veamos un posible proceso de generación de un individuo:

número aleatorio	posición seleccionada	genotipo
2	1, **2**,3,4,5	2
3	1,3, **4**,5	2-4
3	1,3, **5**	2-4-5
1	**1**,3	2-4-5-1
1	**3**	2-4-5-1-3

2.4 Función de adaptación

La adaptación de un individuo es la cantidad de material que se necesita para realizar el patrón de cortado que representa. Una posibilidad es considerar hojas de material utilizadas completas. Aquí vamos a refinar la cantidad considerando las hojas completas utilizadas más la porción utilizada de la última hoja. Concretamente, el objetivo es minimizar el número S(x) de hojas de material que requiere una solución x.

Utilizaremos una función objetivo refinada que considera la última hoja sólo parcialmente:

$$F(x) = S(x) - \frac{L - c_L}{L}$$

Donde c_L representa la posición de la última tira de la última hoja. Esta función refinada permite discriminar mejor la calidad de las soluciones que requieren el mismo número de hojas.

Para calcular la adaptación es necesario construir el patrón de cortado: una lista de hojas, cada una de las cuales consta de una lista de tiras, cada una de ellas con una lista de pilas, que constan de una lista de elementos.

```
tipo TPila: vector de TElem;

tipo TTira: vector de TPila;

tipo THoja: vector de TTira;

tipo TCortado: vector de THoja;
```

Para rellenar los datos del cortado correspondiente a un individuo partimos de la lista de elementos especificada en sus genes. Como la representación adoptada para el individuo es una lista de tipos de elementos en lugar de los elementos propiamente dichos, por comodidad construimos la lista de elementos antes de calcular la adaptación. En caso de haber adoptado la representación alternativa por elementos, una mejora al calcular la adaptación es que al ir a colocar un elemento, si no hay espacio, se prueba a rotarlo 90º.

Una vez construida la lista de los elementos, a partir de la que podemos acceder a las dimensiones de cada uno de ellos, construimos el patrón de cortado mediante el siguiente procedimiento:

- Se van colocando elementos en la hoja de material en curso, siguiendo el orden especificado en los genes del individuo.

- El primer elemento colocado (esquina superior izquierda) determina la anchura de la primera pila de la primera tira de la hoja. La altura de la tira inicial sólo está limitada por el tamaño de la hoja.

- Siempre que quepan en la tira en curso, y que los siguientes elementos sean iguales al anterior, se van colocando en la misma pila.

- Si no hay espacio en la pila (la tira puede acabarse), o si tenemos que colocar un elemento de dimensiones distintas a las del anterior, se abre una nueva pila. Si hay espacio suficiente para las dimensiones del elemento, dentro de la misma tira. Si no, se intenta abrir una nueva tira en la misma hoja. Si el espacio es insuficiente se abre una nueva hoja.

- Cada vez que se abre un nuevo elemento, se meten los datos de los que se acaban de cerrar (pila y/o tira y/o hoja) en el patrón de cortado que se está construyendo. Esto también se hace al terminar de colocar los elementos de la lista.

A partir del cortado generado se contabiliza el número de hojas utilizadas, excepto para la última, para la que se calcula la porción correspondiente a las tiras utilizadas de ella.

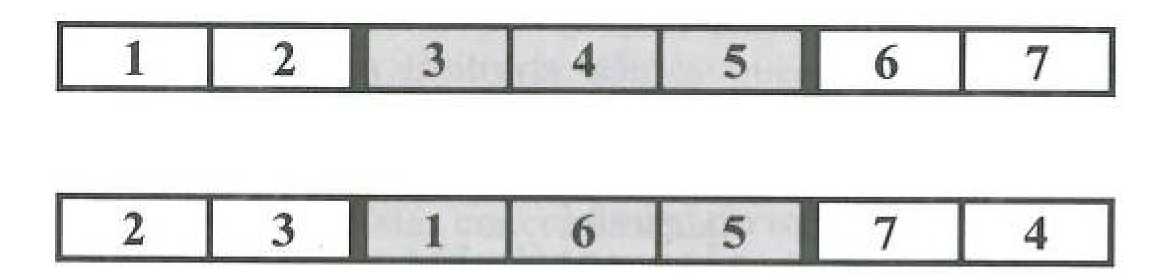
2.5 Operador de cruce

Necesitamos un operador de cruce que dé lugar a permutaciones de los genes de los individuos. Por ejemplo, el cruce monopunto clásico de los algoritmos genéticos no sería válido, ya que produciría individuos en los que podrían faltar elementos y tener otros repetidos.

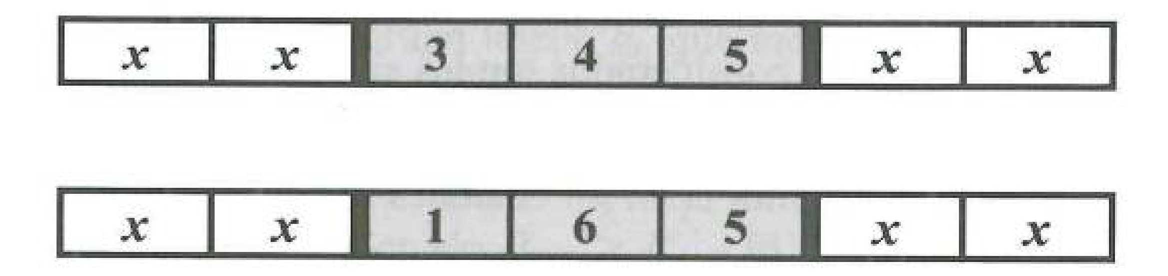
Hay que tener en cuenta que en esta aplicación el orden relativo en el que se colocan los elementos influye en el resultado del cortado. Por ello, parece indicado utilizar algún tipo de cruce que tienda a preservar el orden de porciones de los individuos cruzados. Uno de estos operadores de cruce es el *cruce por orden*.

En el cruce por orden se eligen dos puntos de cruce, se intercambian los segmentos determinados por dichos puntos y se completan los hijos con genes del otro progenitor que no se repitan y tomados a partir de uno de los puntos de corte. Por ejemplo:

1. Se eligen al azar los puntos de cruce que determinan los segmentos a intercambiar:

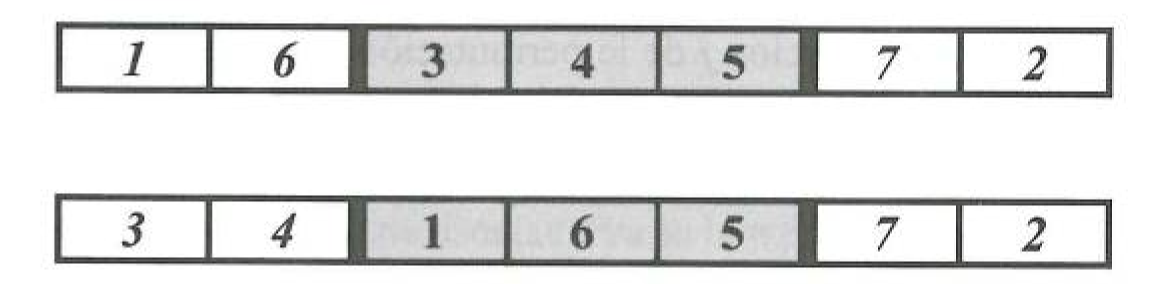


2. Se seleccionan los segmentos:



3. Las posiciones pendientes de especificar se toman del padre al que no pertenece el segmento, partiendo de uno de los puntos de cruce y copiando los elementos de forma que se conserva el orden relativo y se ignoran los que ya están presentes en ese hijo. Al llegar al final de la secuencia se continúa por el principio.

En el ejemplo presente, y partiendo del segundo punto de cruce en ambos padres, se obtienen los siguientes individuos:



En caso de haber adoptado la representación por elementos independientes, un cruce alternativo que favorece el agrupamiento de elementos iguales es el *Cruce de grupos por orden* (variante del cruce por orden): se eligen dos puntos de cruce aleatoriamente, pero al menos alguno de ellos debe separar posiciones de distinto tipo en ambos individuos.

2.6 Operador de mutación

Se consideran dos operadores de mutación, seleccionando en cada mutación uno de ellos al azar:

```
para cada i desde 0 hasta tam_pob hacer{
    prob = alea();
    si (prob < prob_mut){
       tipo_mut = alea_ent(0,1);
       si tipo_mut = TIPO_MUT_PUNTUAL:
          ...
       si tipo_mut = TIPO_MUT_GRUPOS:
          ...
```

Los tipos de mutación considerados son los siguientes:

- *Intercambio puntual*: elige dos posiciones aleatoriamente y las intercambia.
- *Intercambio de bloques*: intercambia dos bloques no solapados elegidos aleatoriamente de longitud l aleatoria $(1 \leq l < m/2)$, siendo m el número de tipos de elementos. De esta forma los bloques cortos se eligen con mayor probabilidad, aunque los grandes que no sobrepasen la mitad de la longitud del genotipo también son posibles.

En caso de haber adoptado la representación por elementos independientes, un operador de mutación que favorece el agrupamiento es la *Mutación de agrupamiento:* se usa para formar secuencias más largas de elementos del mismo tipo. Se elige al azar una posición j de la permutación π de elementos. Entonces se recorre la permutación y cada elemento del mismo tipo que π_j se incorpora a la secuencia alrededor de j con una probabilidad elegida aleatoriamente de [0,1).

2.7 Parámetros del algoritmo

Tamaños de población superiores a 1000 y número de iteraciones en función de los datos de entrada. Para realizar un estudio de los parámetros más adecuados para el algoritmo proponemos implementar un sistema que permita variar los parámetros (tamaño de población, etc.) interactivamente.

3. VARIANTES DEL DISEÑO DEL ALGORITMO

Podemos considerar restricciones adicionales del proceso de producción. Un robot situado al final de la cadena de cortado coloca los elementos terminados

en vagones situados en tres terminales diferentes. Los elementos se reparten en grupos lógicos de acuerdo con las especificaciones del cliente, y cada vagón sólo puede transportar elementos del mismo grupo lógico. Puesto que los vagones no se pueden intercambiar de forma arbitraria, sólo se pueden producir simultáneamente elementos de un máximo de tres grupos lógicos. La solución a nuestro problema debe proporcionar también una planificación del proceso de carga en los vagones de los elementos terminados. Más concretamente:

- Las hojas de material se caracterizan también por su grosor.
- Cada tipo de elemento E_i, además de los otros datos que lo caracterizan, tiene asignado un grupo lógico g_i, que indica el cliente al que se deben enviar los elementos.
- Los vagones tienen asignado un nivel mínimo de llenado $M_{min} > 0$ y un nivel máximo de llenado $B_{max} > B_{min}$. El nivel de llenado de un vagón viene dado por el grosor acumulado de los elementos que contiene.
- Los elementos terminados se cargan en los vagones en el mismo orden en que se producen. Inicialmente hay tres vagones vacíos colocados en los tres terminales disponibles.
- El primer elemento que se coloca en un vagón inicializa el grupo lógico del vagón, es decir, decide su dirección de destino. A partir de ese momento sólo se pueden colocar en el vagón elementos del mismo grupo lógico.
- Un vagón se cierra de forma obligatoria y se reemplaza por uno nuevo, vacío, si su nivel de llenado alcanza B_{max}, o si todos los elementos del grupo lógico que tiene asociado ya se han producido y cargado.
- Un vagón se puede cerrar y reemplazar antes si su nivel de llenado es al menos B_{min}.
- Sólo se consideran válidos los patrones de cortado para los que exista una planificación de la carga que cumpla los requisitos anteriores.

Ahora, el formato de los ficheros con los datos de prueba es el siguiente:

ANCHURA ALTURA GROSOR NIVEL_MIN NIVEL_MAX (material)

GRUPO_LOG

ANCHURA ALTURA CANTIDAD (de un elemento requerido)

GRUPO_LOG

ANCHURA ALTURA CANTIDAD (de un elemento requerido)

...

donde la primera fila especifica las características de las hojas de material (incluyendo el grosor, que ahora debe tenerse en cuenta para la carga de los vagones), y el nivel mínimo y máximo de los vagones. Las siguientes líneas describen los grupos lógicos y las órdenes asociadas a ellos: las líneas con un solo número empiezan un grupo lógico (con el identificativo del grupo). Las líneas por debajo de ella describen las órdenes asociadas a ese grupo, hasta que una nueva línea con un único número empieza un nuevo grupo, o el fichero termina.

3.1 Diseño e implementación

Para la implementación es necesario añadir la nueva información a la representación de los datos. A la representación de la hoja de material hay que añadir el grosor de las hojas. También podemos añadir aquí los niveles máximo y mínimo de llenado de los vagones, o almacenarlos en otro tipo de dato separado.

```
tipo TForma = registro{ // hojas de material
    real ancho;
    real alto;
    real grosor;
    real nivel_max;
    real nivel_min;
}
```

Los tipos de los elementos y los elementos deben incluir el grupo lógico al que pertenecen.

```
tipo TTipo_elem = registro{
    real ancho;
    real alto;
    entero cantidad;
    entero grupo_logico;
}

tipo TElem = registro{
    real ancho;
    real alto;
    entero grupo_logico;
```

```
}
```

Se introduce un nuevo tipo de dato para representar los vagones, que incluye el terminal asignado al vagón, su nivel de llenado actual, el grupo lógico de los elementos que está cargando y la lista de elementos que constituye su carga:

```
tipo TVagon = registro{
   entero terminal;
   entero grupo_logico;
   real nivel;
   TLElem carga;
}
```

Representamos también la lista de vagones:

```
tipo TVagones: vector de TVagon;
```

Ahora el individuo también incluye el vector de vagones:

```
tipo TIndividuo = registro{
   TGenes genes; // cadena de enteros (genotipo)
   real aptitud; // función de evaluación
   real puntuacion; //puntuación relativa:
    adaptación/sumadaptación
   real punt_acu; // puntuación acumulada para sorteos
   TCortado cortado; // fenotipo
   TVagones vagones;
};
```

La forma de decodificación para el cálculo de la aptitud en este caso consiste en tomar elementos de la lista para los que existe un vagón disponible, pasando al siguiente elemento si no lo hay. El proceso continúa hasta que se hayan producido todos los elementos de la lista. Para buscar un terminal válido para un elemento, se considera el grupo lógico del elemento g_i, y el estado actual de los vagones de los tres terminales de carga. Si g_i aparece como grupo lógico de uno de los vagones de los terminales, entonces ese es un terminal válido. Sino, buscamos los terminales cuyo vagón está o bien vacío o ha alcanzado su nivel de llenado mínimo B_{min}. En el último caso, el vagón se cierra y se reemplaza por uno nuevo. Si se puede encontrar un terminal válido, se almacena su identificador como el destino del elemento, formando parte de la solución.

Proyecto 6

CONTROL DE TRÁFICO AÉREO

1. DESCRIPCIÓN DEL PROBLEMA

En este proyecto se desea implementar un algoritmo evolutivo para planificar el tráfico de llegadas de vuelos en un aeropuerto. El problema está basado en un artículo científico de Cheng y colaboradores [CHE99], cuyos experimentos se desarrollan aquí con detalle. Concretamente, el sistema debe asignar pistas de aterrizaje y un orden a las llegadas de *n* vuelos a *m* pistas de aterrizaje. Los datos de partida son los siguientes:

- Para cada uno de los *n* vuelos, se dispone de un tiempo estimado de llegada (TEL) a cada una de las *m* pistas disponibles. Se trata del tiempo mínimo para alcanzar la pista. Se supone que todas las posibles restricciones sobre el vuelo (condiciones climáticas, potencia de motores, etc.) están incluidas en estos valores TEL. El tiempo planificado al asignar un vuelo a una pista no puede ser anterior que el correspondiente TEL.

- Todos los vuelos que se asignan a una misma pista tienen que cumplir reglas de separación específicas entre aviones, en base al tipo de los aviones. Se consideran tres tipos de aviones: Pesado (W), Grande (G), Pequeño (P). El tiempo de separación depende de la pista y del tipo de avión, y es un dato de entrada (SEP).

- Cuando se asigna un vuelo a una pista, el retardo resultante se define como la diferencia entre el tiempo de llegada asignado (TLA) y el menor TEL de ese vuelo en todas las pistas.

Utilizaremos, entre otros, los siguientes datos de prueba:

Identificadores de vuelo **IV**: **(UA138, UA532, UA599, NW358, UA2987, AA128, UA1482, NW357, AA129, UA2408, UA805, AA309)**;

para los que los correspondientes tipos de avión (TV) vienen dados por la lista:

TV = (W,G,W,W,P,W,G,W,W,P,W,G);

La lista de TELs para los 12 vuelos y las 3 pistas viene dada por la siguiente matriz:

	*V*1	*V*2	*V*3	*V*4	*V*5	*V*6	*V*7	*V*8	*V*9	*V*10	*V*11	*V*12
*P*1	11	15	6	6	9	7	15	6	6	9	7	9
*P*2	10	17	7	7	12	6	17	7	7	12	6	7
*P*3	9	19	8	8	15	5	19	8	8	15	5	5

Toda secuencia de aterrizaje debe satisfacer los requisitos de separación entre vuelos, que se especifican por la matriz SEP. La matriz usada en el problema de prueba es:

	W	*G*	*P*
W	1	1.5	2
G	1	1.5	1.5
P	1	1	1

donde las filas representan el tipo del avión que llega, y las columnas representan al avión anterior. Por ejemplo, un avión pequeño (P) que sigue a uno pesado (W) requiere dos unidades de tiempo de separación, mientras que un avión grande (G) que sigue a uno pequeño (P) sólo requiere una unidad de tiempo.

2. DISEÑO DEL ALGORITMO

Se trata de un problema de optimización con restricciones. En este caso existe una codificación sencilla que nos permite garantizar que todos los individuos cumplen las restricciones, por lo que no necesitamos introducir penalizaciones por la violación de restricciones.

Vamos a comenzar por el diseño de la representación de los datos de entrada. Representamos el número y tipos diferentes de aviones que pueden llegar para aterrizar:

```
constante entero num_tipos = 3
tipo enumerado TTipo_avion = (pesado, grande pequeño);
```

Introducimos la estructura *TVuelo* para representar la información que tenemos de cada vuelo: su nombre, tipo y tiempo estimado de llegada a cada pista. Para registrar esta última información se introduce el tipo *TTel_vuelo*. La lista con todos los vuelos está representada por el tipo *TLVuelos*. Es la posición en esta lista lo que codifican los individuos del algoritmo.

```
tipo TTel_vuelo: vector de real;
// tiempo estimado de llegada por pista

tipo TVuelo = registro{
   cadena_caracteres nombre; // nombre del vuelo
   TTipo_avion tipo; // tipo del avión
   TTel_vuelo tel_vuelo; // TEL por pista
}

tipo TLVuelos: vector de TVuelo; // lista de vuelos
```

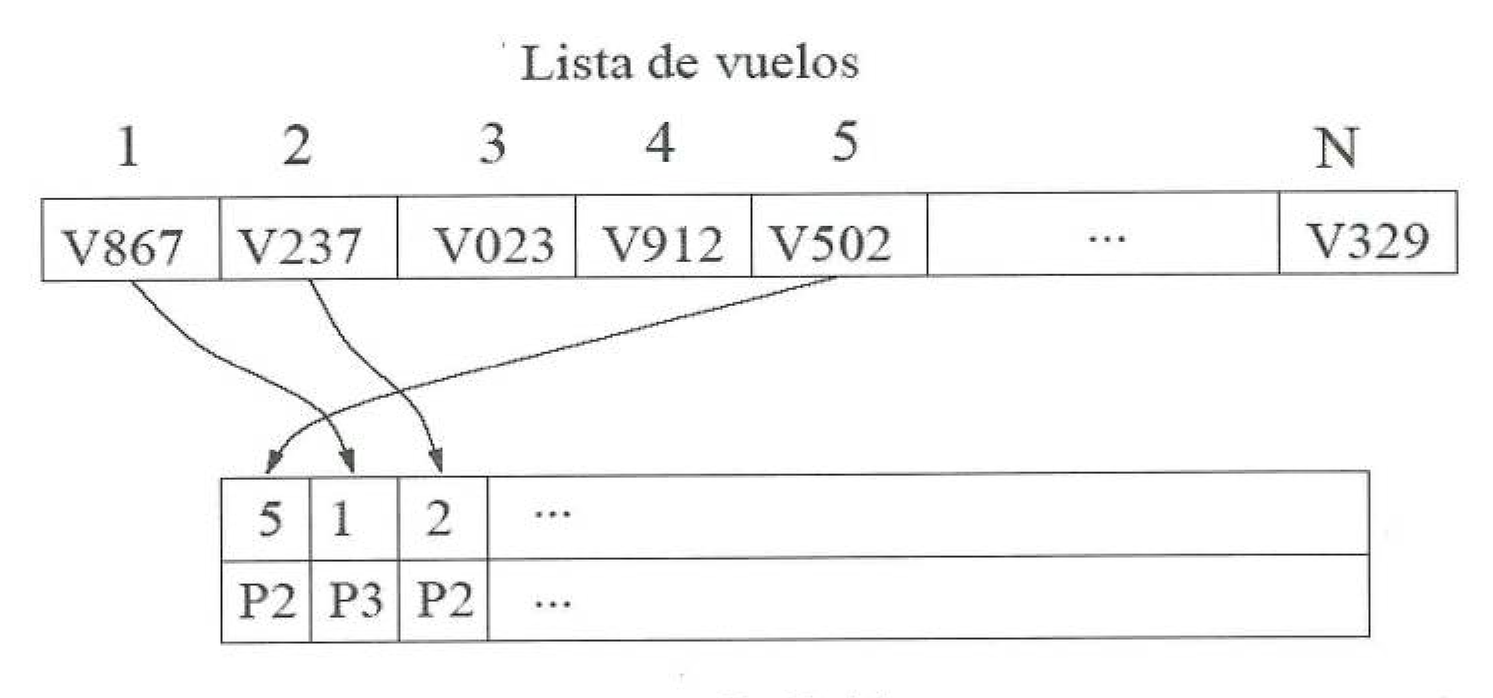
Utilizamos también una estructura de datos para almacenar la especificación de separación entre vuelos:

```
tipo TSep_tipo: vector de real; // separación por tipo
tipo TMSep: vector de TSep_tipo; // matriz de separaciones
```

2.1 Representación de los individuos

Cada individuo se representa por una secuencia de n genes, siendo n el número de vuelos. Cada gen consta de dos valores, el vuelo (o su posición en la lista de entrada) y la pista asignada. La secuencia de los valores de vuelo de los

genes define el orden relativo de los vuelos, y conjuntamente con la secuencia de pistas, determinan la asignación de pistas, y los tiempos de llegada de los vuelos, que se calculan en la función de adaptación. El orden relativo de los vuelos sólo es relevante para vuelos asignados a la misma pista. El tiempo de llegada de los vuelos se determina seleccionando el menor TLA posible en la pista asignada. Los requisitos de separación entre vuelos sucesivos se tienen en cuenta al determinar el TLA durante el cálculo de la función de adaptación.



Para representar a los individuos seguimos utilizando una cadena de genes, es decir, podemos utilizar una estructura similar a la de los algoritmos genéticos:

```
tipo TIndividuo = registro{
    TGenes genes; // cadena de genes (genotipo)
    real x; // fenotipo
    real adaptación; // función de evaluación
    real puntuacion;
    //puntuación relativa: adaptación/sumadaptación

   real punt_acu; // puntuación acumulada para sorteos
};
```

pero ahora la cadena de genes no es una cadena de booleanos, sino que cada gen tiene la estructura necesaria para almacenar los datos de un vuelo: *pos_vuelo*, *pista* y *hora*. El campo *pos_vuelo* es la posición que ocupa el vuelo correspondiente en la lista de vuelos. A partir de esta posición podemos acceder a información sobre el vuelo, evitando tener repetida la información en todos los individuos. *pista* es el número de pista asignada al vuelo para aterrizar. *hora* es la hora resultante de la

asignación de pistas que hace el individuo calculada en la función de adaptación. Resulta útil dejarla almacenada para depuración y visualización del resultado final.

```
tipo TGenes : vector de TGen;
tipo Tgen = registro{
   entero pos_vuelo; // posición en la lista de vuelos
   entero pista; // número de la pista asignada
   real hora; // hora de aterrizaje resultante
}
```

2.2 Generación de la población inicial

Para crear la población inicial generamos una secuencia aleatoria de vuelos y a cada uno se le asigna aleatoriamente una de las posibles pistas. Para construir una secuencia aleatoria de vuelos sin repeticiones construimos una lista inicial de vuelos de la que se va seleccionando y borrando el siguiente de la secuencia. Si tuviéramos 5 vuelos, un posible proceso de generación de la secuencia de vuelos de un individuo sería:

número aleatorio	posición seleccionada	secuencia de vuelos
2	1, **2**,3,4,5	2
3	1,3, **4**,5	2-4
3	1,3, **5**	2-4-5
1	**1**,3	2-4-5-1
1	**3**	2-4-5-1-3

2.3 Función de adaptación

El cálculo de la adaptación requiere generar la planificación de vuelos a partir de la secuencia de vuelos y pistas del individuo. La adaptación de un individuo se define como la suma del cuadrado de todos los retardos, y es un valor a **minimizar**. Para calcular la adaptación necesitamos ver qué hora de aterrizaje le corresponde a cada vuelo de acuerdo con el orden de la secuencia de vuelos del individuo, las pistas asignadas y las restricciones del problema.

Para ver el resultado de aplicar las restricciones del problema vamos a representar todas las pistas consideradas en el problema, y a ir asignando vuelos en el orden especificado por el individuo y cumpliendo las restricciones del problema.

Por tanto el primer paso de la función de adaptación consiste en preparar un vector de pistas, en cada una de las cuales se colocan los vuelos asignados a ella, ordenados por su TEL.

Introducimos una representación de una pista y de una lista de pistas para representar la ordenación de vuelos por pista a la que da lugar un individuo.

```
tipo TAsignacion = registro{
   real tel; // tel del vuelo
   entero pos_gen; // posición del gen
}
// lista de asignaciones a la pista ordenadas por TEL
tipo TPista: lista ordenada de TAsignacion;
tipo TPistas: vector de TPista; // lista de pistas
```

Utilizando una forma de representar las pistas como la anterior o similar, podemos implementar la función que calcula la adaptación de un individuo. A continuación presentamos un esquema del cálculo.

```
funcion adaptacion(TIndividuo individuo, entero lcrom,
        TLVuelos lvuelos, TMSep sep, entero num_pistas)
{
   TPista pista; // una de las pistas
   TPistas pistas; // vector de pistas
   real adapt, hora, tel,tla,menor_tel;

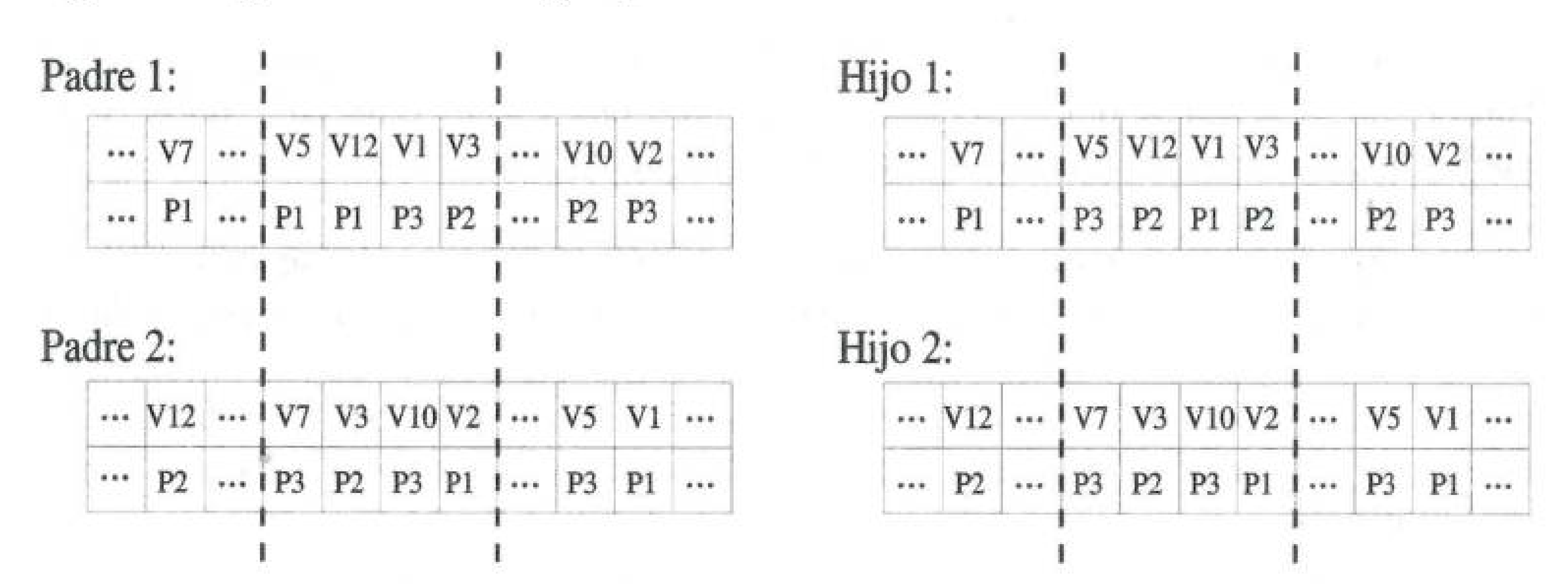
   // se prepara el vector de pistas:
   // en cada pista se colocan los vuelos asignados
   // a ella ordenados por su TEL
   { ... }
   // se calcula la adaptación
   adapt = 0;
   para cada pista de pistas hacer{
    para cada vuelo de la pista hacer{
     si es el primer vuelo de la pista entonces
          hora = tel(pista, vuelo);
     eoc{
      tla = hora asignada al vuelo anterior de la pista;
      tla = tla + separacion(tipo_vuelo_anterior,
              tipo_vuelo_actual);
      si tla >= tel(pista, vuelo) entonces
             hora = tla;
      eoc
             hora = tel(pista, vuelo);
         }
      }
   }
   // se calcula el menor_tel del vuelo en cualquier pista
```

```
                    { ··· }
    adapt = adapt + (hora - menor_tel)^2
    devolver adapt;
}
```

La primera parte consiste en preparar las pistas. Para ello se consulta en los genes del individuo la asignación que hace de pistas a vuelos, y se coloca en cada pista los vuelos especificados en el individuo, pero ordenados por su TEL. Después se calcula la adaptación: para cada vuelo de cada pista se calcula la hora de aterrizaje de acuerdo con la asignación de pistas del individuo, y se añade a la adaptación el cuadrado de la distancia del mejor tiempo posible de ese vuelo en la mejor pista.

2.4 Operador de cruce

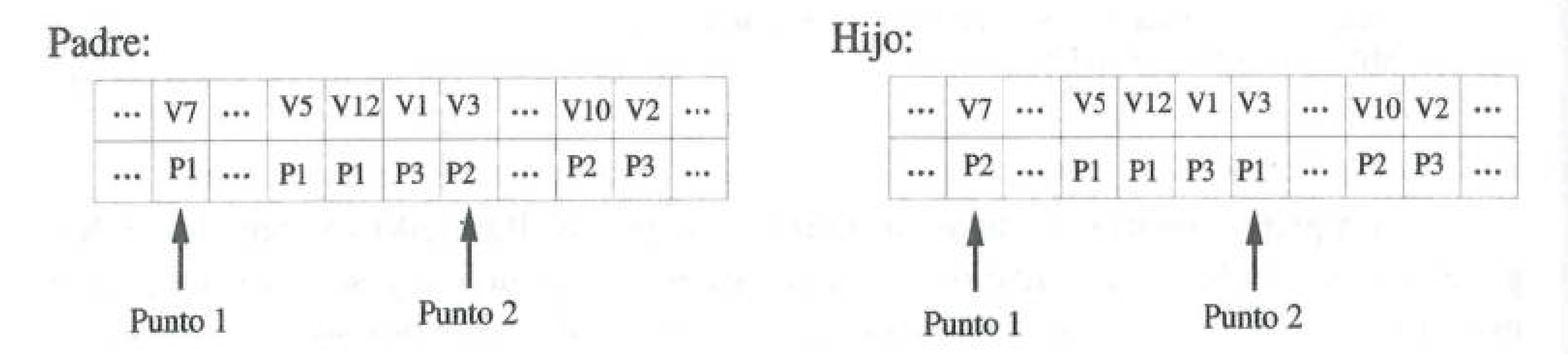
Se seleccionan dos puntos de cruce que determinan un segmento y se intercambian entre ambos padres las pistas asignadas a los vuelos del segmento. La siguiente figura muestra un ejemplo:



En el ejemplo vemos como el hijo 1 es una copia del padre 1, excepto en que la pista asignada a los vuelos V5, V12, V1, y V3, que están entre los puntos de corte, se toman de la especificación del padre 2. La construcción del hijo 2 es análoga, en este caso partiendo del padre 2.

2.5 Operador de mutación

Consiste en cualquier permutación de las pistas asignadas a dos vuelos. La siguiente figura muestra un ejemplo:



En el ejemplo vemos como se han seleccionado dos puntos de mutación al azar, en este caso los genes correspondientes a los vuelos V7 y V3. En el individuo hijo resultante de la mutación las pistas que tengan asignadas estos vuelos en el padre se han intercambiado.

2.6 Consideraciones adicionales

El valor óptimo para los datos dados como ejemplo que se obtiene utilizando la función de adaptación que hemos definido es de 11.25.

Sin embargo, el algoritmo debe funcionar también, lo que se aconseja comprobar, con otros datos de entrada, aunque estos sean más complejos: más vuelos, para el mismo número de pistas o para muchas más pistas.

Para comprobar que el algoritmo funciona correctamente se debe buscar una forma de ir presentando por pantalla la asignación de vuelos a pistas y la hora que se asigna a cada vuelo. Esta información resultará muy útil en la fase de depuración y ajuste de parámetros del algoritmo. En cualquier caso, al terminar la ejecución el resultado final debe presentase de esta forma, para que sea inteligible para el usuario.

Valores de partida de los parámetros para iniciar el estudio de los más adecuados pueden ser un tamaño de población de 100 individuos, un número máximo de 100 generaciones, una tasa de cruce del 40% y una tasa de mutación del 5%.

3. UNA REPRESENTACIÓN ALTERNATIVA

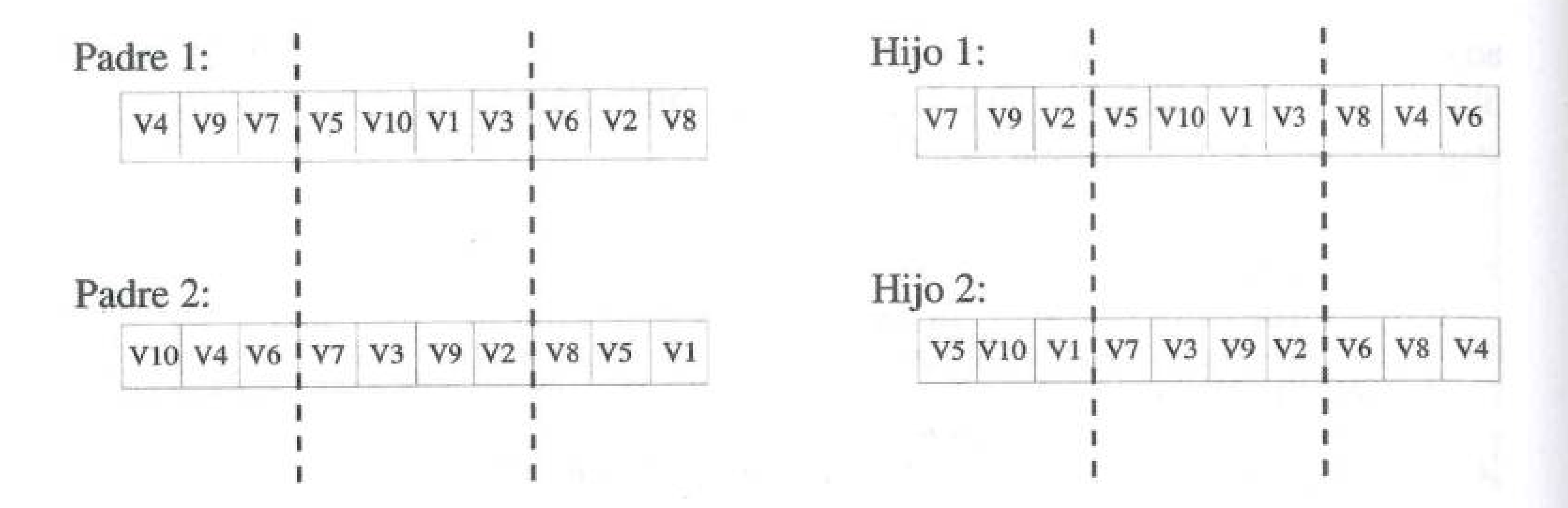
Algoritmo evolutivo alternativo. En este esquema un individuo se representa por una secuencia de genes que constan de un único valor. Cada individuo representa una lista de prioridad entre vuelos: avanzando por la lista, se asigna a cada vuelo la pista y el tiempo que minimicen su TLA, y así su retardo. Los requisitos de separación se tienen en cuenta al calcular su TLA. La función de

adaptación se define como en el caso anterior, aunque la forma de generar la planificación a partir del individuo es específica para esta representación. Un esquema de la función puede ser el siguiente:

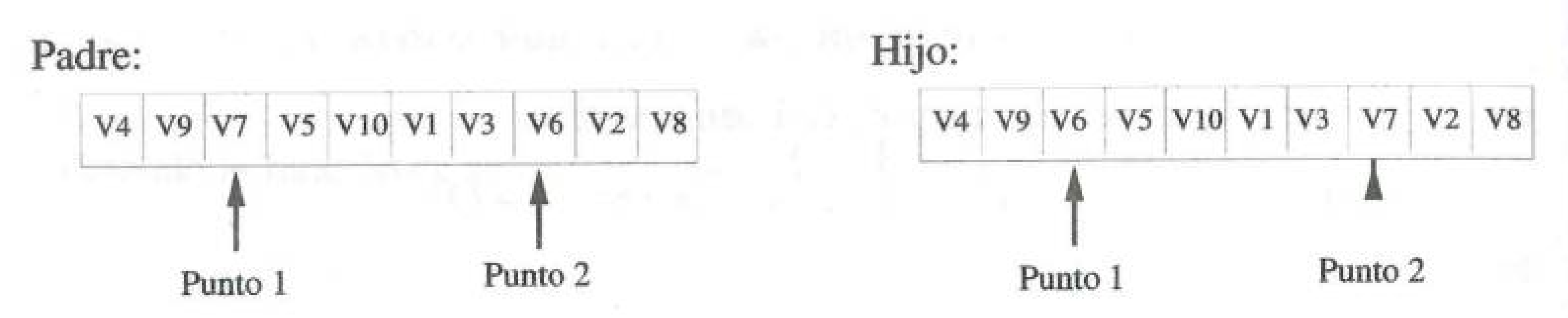
```
adapt = 0;
para cada vuelo del individuo hacer
   // se le asigna la pista que minimiza su TLA
   menor_tla = ∞ ;
   para cada pista hacer {
      sep = separacion(tipo_vuelo_anterior,
    tipo_vuelo_actual);
      tla = máximo(tla(vuelo_anterior)+ sep, tel);
      // se actualiza el menor tla y la pista
      // asignada al vuelo si corresponde
                                   {···}
   }
   // se coloca el vuelo en la pista que minimiza el tla
                             {···}
   // se calcula el menor_tel del vuelo
                             {···}
   adapt = adapt + (menor_tla - menor_tel)²
```

Siguiendo el orden de vuelos especificado en el individuo, se va asignando a cada vuelo la pista que minimiza su TLA, de acuerdo con la asignación de vuelos a pistas realizada hasta el momento. Se calcula también el menor TEL del vuelo considerado, es decir, el TEL menor de ese vuelo de entre todas las pistas. Por cada vuelo la adaptación se va incrementando en el cuadrado de la diferencia entre el menor TLA y el menor TEL.

Los operadores genéticos se ajustan a la nueva representación. Ahora el cruce intercambia un segmento de la secuencia de vuelos de cada padre entre dos puntos elegidos aleatoriamente. Hay que garantizar que las secuencias de vuelos de los hijos resultantes del cruce sigan siendo permutaciones de los vuelos: no puede haber repeticiones ni ausencias. Un operador que nos sirve en este caso es el *cruce por orden*. En este cruce cada hijo se construye partiendo del segmento definido por dos puntos de cruce en uno de los progenitores. Para los genes restantes se toman los valores del otro padre, siguiendo la secuencia a partir de uno de los puntos de cruce y excluyendo los valores repetidos. La siguiente figura muestra un ejemplo, cuando se parte del segundo punto de corte para completar las secuencias de los hijos:



La mutación intercambia los vuelos de dos posiciones de mutación elegidas aleatoriamente. De esta forma el individuo resultante sigue siendo una permutación de los vuelos. La siguiente figura muestra un ejemplo:



Es interesante comparar los resultados obtenidos con este esquema con los de la representación anterior, y estudiar los parámetros más adecuados para el algoritmo en cada caso.

4. CON RESTRICCIONES ADICIONALES

Podemos considerar restricciones adicionales: una preferencia por asignar los vuelos de un tipo a una determinada pista, por excluir de una pista vuelos de un determinado tipo, o por asignar dos vuelos determinados a distintas pistas, o a la misma. Podemos considerarlas como restricciones sobre las que hay flexibilidad, es decir, como preferencias más que como requisitos inviolables.

Las restricciones inherentes al problema (separación entre vuelos, etc.) las hemos tratado con un mecanismo de codificación que garantiza que nunca se generan individuos que las violen. Las restricciones flexibles que introducimos en este apartado las podemos tratar con un mecanismo de penalización. Concretamente sumaremos a la adaptación una cantidad, específica por cada tipo de restricción, por cada restricción violada.

Algunos ejemplos de restricciones que podemos considerar son los siguientes:

- REST_INCLUIR: para especificar que los aviones de un determinado tipo sólo aterricen en una determinada pista.
- REST_EXCLUIR: para especificar que los aviones de un determinado tipo no deben aterrizar en una determinada pista.
- REST_JUNTOS: para especificar que dos vuelos determinados aterricen en la misma pista.
- REST_SEPARAR: para especificar que dos vuelos determinados aterricen en distintas pistas.

Un ejemplo de valores de penalización que se pueden utilizar para los distintos tipos de restricciones son los siguientes:

REST_INCLUIR 2

REST_EXCLUIR 1

REST_JUNTOS 3

REST_SEPARAR 3

Las restricciones, que son datos de entrada, se almacenan en estructuras de datos específicas. Por ejemplo, para la restricción de tipo REST_EXCLUIR, podríamos utilizar las siguientes estructuras:

```
tipo TLRest_excluir = registro{
   TTipo_avion tipo_avion; // tipo de avión que especifica
                           // la rest.
   entero pista; // pista de la que se excluye
}

tipo TLRest_excluir : vector de TRest_excluir;
```

Al calcular la adaptación se deben tener en cuenta las restricciones:

- Para las restricciones de tipo *REST_INCLUIR* se comprueba en todas las pistas distintas a la especificada en la restricción que no hay vuelos del tipo especificado, penalizándolos si los hay.

- Para las restricciones de tipo *REST_EXCLUIR* se recorre la pista especificada en la restricción y se penaliza cada vuelo del tipo especificado que se encuentre.

- Para las restricciones de tipo *REST_JUNTOS* se localiza la pista de los dos vuelos especificados en la restricción y se penaliza si son distintas.

- Para las restricciones de tipo *REST_SEPARAR* se localiza la pista de los dos vuelos especificados en la restricción y se penaliza si son iguales.

Es interesante estudiar las soluciones que proporciona el sistema para distintas asignaciones de penalización a cada tipo de restricción.

Proyecto 7

RESOLUCIÓN DEL SUDOKU

1. DESCRIPCIÓN DEL PROBLEMA

El Sudoku es un pasatiempo que se popularizó en Japón en 1986 y se dio a conocer en el ámbito internacional en el año 2005. Consiste en un tablero con algunas casillas ocupadas por números y con algunas casillas libres. El pasatiempo consiste en intentar rellenar todas las casillas libres con números enteros respetando una serie de restricciones. Más adelante se explica detalladamente la versión básica del juego.

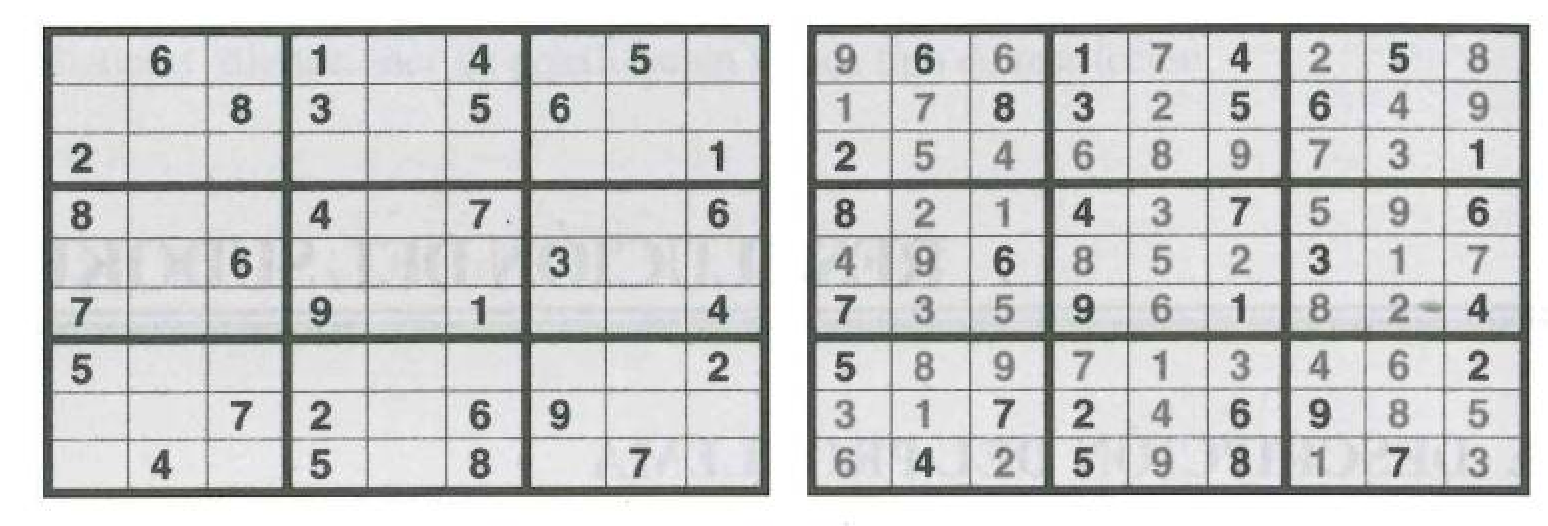
La resolución del Sudoku se ha llevado a cabo utilizando diferentes estrategias, entre las que podríamos destacar los clásicos algoritmos de vuelta atrás y el de ramificación y poda. El objetivo de este proyecto es utilizar un algoritmo evolutivo para resolver el pasatiempo del Sudoku con tableros de 9x9 casillas. Ahora podremos comprobar si un algoritmo evolutivo es una buena estrategia para resolver dicho juego.

Para resolver el problema de Sudoku con un algoritmo genético, como en la mayoría de los AGs, debemos definir tres elementos fundamentales: la representación de los individuos, la función de adaptación y los operadores de transformación.

En la resolución de este problema plantearemos una nueva representación de los individuos, la inclusión de varios operadores genéticos de cruce y mutación basados en problemas de permutaciones y alguna propuesta de función de adaptación que sirve de base para seguir investigando sobre mejoras a la resolución de este problema.

El Sudoku es un pasatiempo que consiste en un tablero de 9 filas x 9 columnas que componen 81 casillas. El tablero está dividido en subcuadrículas de 3x3 casillas. El objetivo del juego es rellenar todas las casillas libres de dicho tablero con los números del 1 al 9 considerando que inicialmente existen algunos números ya colocados en algunas casillas.

En el tablero solución se debe cumplir la restricción de que no se repite ningún número en una misma fila, columna o subcuadrícula. La solución es única. En la figura 1 se muestra un ejemplo de la posible configuración inicial del tablero junto con su solución.



	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6			3			
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

9	6	6	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Figura 1. *Configuración inicial de un tablero y su solución*

El Sudoku tiene una solución única, lo que significa que no se consideran soluciones válidas los tableros con situaciones cercanas a la solución. En este sentido, el Sudoku no parece un problema adecuado para resolver utilizando un algoritmo genético, pero sí es un problema muy útil para experimentar con diferentes operadores genéticos y con diferentes parámetros.

El Sudoku puede considerarse un problema de optimización con restricciones, a nivel de fila, columna y subcuadrícula. A continuación se analiza la representación utilizada para las posibles soluciones, la función de adaptación y los operadores genéticos apropiados al problema.

2. DISEÑO DEL ALGORITMO

El objetivo de nuestro algoritmo es obtener la configuración del tablero que representa la solución del Sudoku. El tablero podemos representarlo de varias formas, pero directamente podemos pensar en una matriz de 9 filas y 9 columnas, que contiene únicamente los valores de las posiciones fijas. Como veremos más adelante la información de un individuo se puede representar de varias formas: ordenando la información por filas, por columnas, por subcuadrículas, etc.

La estructura inicial del tablero en la que las posiciones libres se representan con el valor 0 se codifica del siguiente modo:

```
Tablero = Matriz de enteros
   {
    {0,0,4,0,0,0,0,9,0},
    {7,0,0,0,6,0,0,0,5},
    {0,9,0,5,4,1,8,7,2},
    {0,0,0,1,8,7,0,4,0},
    {2,4,3,6,0,5,1,8,7},
    {0,8,0,3,2,4,0,0,0},
    {9,2,1,8,7,6,0,5,0},
    {6,0,0,0,1,0,0,0,8},
    {0,3,0,0,0,0,7,0,0}
   };
```

El algoritmo utilizado sigue el mismo esquema del algoritmo genético básico pero modificando los operadores genéticos para adaptarlos a la representación asociada al problema.

```
  evaluación poblacion;
  para cada generación desde 1 a MAXGEN hacer {
      seleccion
      reproduccion
      mutacion
      evaluación poblacion;
  }

devolver mejorIndividuo;
```

Este es el esquema más simple que hemos visto, aunque también podemos incluir el esquema con elitismo:

```
evaluación poblacion;
para cada generación desde 1 a MAXGEN hacer {
      separa elite
      seleccion
      reproduccion
      mutacion
      incluye elite
      evaluación poblacion;
  }

devolver mejorIndividuo;
```

2.1 Representación de los individuos y la población

Los individuos deben estar formados, además de por los datos que se requieran para el algoritmo evolutivo, por una matriz que representa el tablero. El individuo o tablero se puede representar de diferentes formas, pero podemos destacar dos:

- **Representación 1**: el cromosoma se representa mediante una lista con las celdas vacías del tablero ordenada por filas, y cada celda puede tener un valor entre 1 y 9. Para el tablero de la figura 1 el cromosoma se representa mediante una lista de 52 números enteros comprendidos entre 1 y 9.

- **Representación 2**: el cromosoma se representa mediante un vector compuesto por 9 genes de 9 elementos cada uno. Cada gen se puede corresponder con una fila, una columna o una subcuadrícula del tablero.

Como veremos más adelante, los operadores de cruce y mutación varían dependiendo del tipo de representación seleccionado. Sabiendo que el tablero original es una matriz, podemos convertir dicha matriz en un vector unidimensional ordenado por filas, por columnas o por cuadrículas.

Si elegimos la representación 2, podemos considerar el cromosoma como un vector de 81 posiciones donde las posiciones fijas del tablero se respetan y las posiciones vacías se representan como 0.

En este proyecto hemos decidido almacenar los valores ordenados por filas de forma que tenemos un acceso más cómodo a las filas y a las columnas. Si suponemos el siguiente tablero :

	6				5			3
		8						
2					1			9
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

Definiremos una estructura correspondiente al cromosoma que permita almacenar los valores fijos del tablero inicial y que respete los huecos donde se pueden colocar valores nuevos.

Siguiendo dicho esquema, el cromosoma correspondiente al tablero anterior sería como el siguiente:

0	6	0	0	0	5	0	0	3	0	0	8	0	0	0	0	0	0	..

Utilizando esta representación el siguiente pseudocódigo define un cromosoma como un vector de genes o filas y cada gen o fila es una vector de celdas o casillas del tablero. Cada casilla contiene el valor numérico y un indicador para saber si es un valor fijo.

```
tipo TIndividuo = registro{

 genes: vector de Gen; //vector de genes o filas
 real adaptación; //función de evaluación
 real puntuacion;
 //puntuación relativa:adaptación/ sumadaptación
 real punt_acu; //puntuación acumulada
 booleano elite; // elitismo
}
```

Cada elemento de tipo Gen representa una fila del Sudoku; en realidad un gen o fila es un vector de casillas:

```
tipo Gen = vector de Tcasilla;
```

Una casilla representa una celda del tablero del siguiente modo:

```
tipo TCasilla = registro{
 entero valor; // el valor numérico de la casilla
 boolean fija; // indica si es una posición fija o no
}
```

La población es una colección de individuos que se representa fácilmente mediante un vector:

```
tipo TPoblacion: vector de TIndividuo;
```

2.2 Generación de la población inicial

Inicialmente cada celda del tablero (cromosoma) tendrá un valor del 0 al 9. Los huecos del cromosoma se inicializan con valores aleatorios entre 1 y 9, respetando las posiciones fijas y sin generar elementos repetidos dentro de la fila, de forma que al calcular la adaptación del individuo no tengamos que preocuparnos por esta restricción.

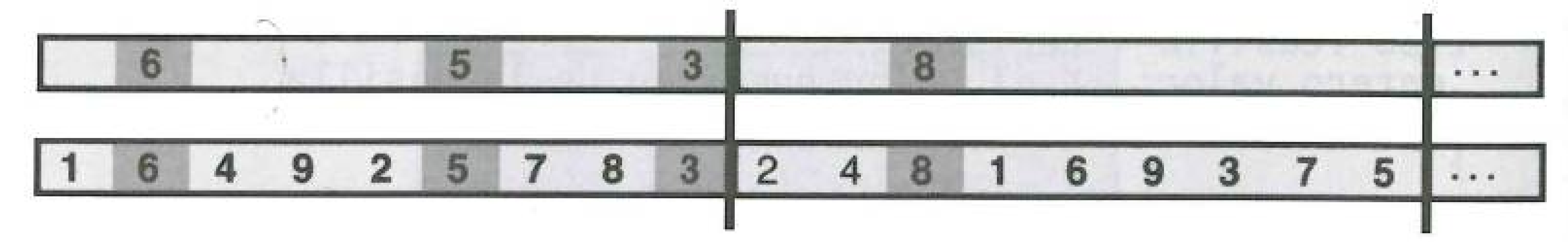
La siguiente función rellena las casillas libres de cada fila con valores numéricos distintos entre 1 y 9, excluyendo los valores colocados en posiciones fijas de la fila:

```
funcion rellenarCasillas {

Lista_de_enteros cambios; //valores por colocar en la fila
Lista_de_Enteros entrada; //valores fijos en la fila

para cada i desde 0 hasta tamaño_fila hacer {
    añadir(cambios,i+1);
    si casilla i es fija entonces
          añadir(entrada, valor de la casilla i);
    }
    Quitar de cambios los elementos de la lista entrada.
    para cada i desde 0 hasta tamaño_fila hacer {
    si casilla i no es fija entonces
          posicion = alea_int(0, tamaño_cambios-1);
          valor = sacar(cambios,posicion);
          Poner valor en casilla i;
      }
}
```

Esta inicialización garantiza que no hay repetición de elementos a nivel de fila, como se puede ver en este ejemplo donde los valores sin sombreado son los valores asignados a las casillas vacías.



2.3 Función de adaptación

La función de adaptación depende del tipo de representación que haya seleccionado. Sugerimos al lector el uso de dos funciones de adaptación de los individuos según el tipo de representación utilizado:

- Función de adaptación por faltas
- Función de adaptación ponderada

Función de adaptación por faltas:

Esta función la utilizamos con el tipo de Representación 1, donde los individuos se representa mediante una lista con los valores de las celdas vacías del tablero ordenada por filas.

Esta primera función de adaptación que podemos usar para medir la adaptación de una solución puede ser muy sencilla: basta con contar el número de faltas de cada fila, columna y subcuadrícula, considerando como falta la repetición de un elemento. Siguiendo este proceso, cada celda puede acumular hasta 3 faltas.

Los elementos de cada fila, columna y cuadrícula deben ser los valores del conjunto de dígitos = {1,2,3,4,5,6,7,8,9}.

La función calcula el número de faltas o elementos incorrectos en cada fila, columna y subcuadrícula:

f(x) = Faltas filas + Faltas columnas + Faltas cuadrículas

Se trata de un problema de minimización y el objetivo a conseguir es adaptación 0. La siguiente función calcula el total de faltas:

```
funcion evaluaFaltas : entero {
    entero numFaltas=0;
    para cada i desde 0 hasta longCromosoma hacer{
      numFaltas = numFaltas + faltasFila(k);
      numFaltas = numFaltas + faltasColumna(k);
      numFaltas = numFaltas + faltasSubMatriz9x9(k);
    }
    devolver numFaltas;
}
```

Función de adaptación ponderada:

Esta función la aplicamos con el tipo de Representación 2, donde los individuos se representan mediante un vector de 9 genes (filas) de 9 elementos cada uno (casillas). Esta función de adaptación está basada en la función propuesta en [MAN06]. Con esta función de adaptación se consideran tres restricciones, cada una de las cuales se penaliza de forma diferente mediante un peso asociado.

- **Restricción de sumas (RS)**: se basa en que en un tablero solución la suma de los valores de cada cuadrícula y cada columna debe ser 45. Se calcula para las columnas y las cuadrículas:

  Restricción de columna *i*: |45 – suma de los valores de la columna *i* |

  Restricción cuadrícula *i* : |45 – suma de los valores de la cuadrícula *i* |

  RS = Suma de restricciones de las 9 columnas + suma de restricciones de las 9 cuadrıculas

- **Restricción de productos (RP):** se basa en que en un tablero solución el producto de los valores de cada cuadrícula y cada columna debe ser 9!.

  Restricción de columna *i*: | 362880 – producto de valores de la columna i |

  Restricción cuadrícula *i* : | 362880 – producto de valores de la cuadrícula i |

  RP = Suma de la raız cuadrada de las restricciones de cada una de las 9 columnas + suma de la raız cuadrada de las restricciones de las 9 cuadrıculas

- **Restricción de elementos ausentes (REA):**

  Restricción de columna *i*: número de valores del conjunto de dígitos del 1 al 9 que faltan en la columna i.

  Restricción de cuadrícula *i*: número de valores del conjunto de dígitos del 1 al 9 que faltan en la cuadrícula i.

  REA = Suma de restricciones de las 9 columnas + suma de restricciones de las 9 cuadrıculas

La mejor función de adaptación obtenida tras numerosos experimentos es la siguiente:

$$F(x) = 5 * RS + RP + 20 * REA$$

La restricción **REA** es la más penalizada al considerar la gravedad de la ausencia de elementos. Partiendo de esta función podemos probar diferentes variaciones de la función de adaptación modificando los pesos de las restricciones y experimentando con dichos cambios.

```
funcion evaluaTablero entero{

//Inicializamos todos los contadores a 0
// . . .

para cada elemento (fila o cuadro) k hacer
{
  sCuadros = sumaCuadro(k);
  gcuadros1 = gCuadros1+ abs(45-sCuadros);
  sColumnas = sumaColumna(k);
  gColumnas1 = gColumnas1 + abs(45-sColumnas);
  }
RS = gFilas1+gColumnas1;

para cada elemento (fila o cuadro) k hacer
{
  pCuadros = productoCuadro(k);
  gcuadros2 = gCuadriculas2+Raiz(abs(362880-pCuadriculas));
  pColumnas = productoColumna(k);
  gColumnas2 = gColumnas2+Raiz(abs(362880-pcolumnas));
  }
RP = gCuadriculas2+gColumnas2;


para cada elemento k hacer
{
  totalFaltasfila = totalFaltasFila + FaltanenFila(k);
  totalFaltasCuadro= totalFaltasCuadro + FaltanenCuadro(k);
  }
REA= totalFaltasfila+ totalFaltasCuadro;

devolver (5*RS +  RP  + 20*REA)

}
```

2.4 Operador de cruce

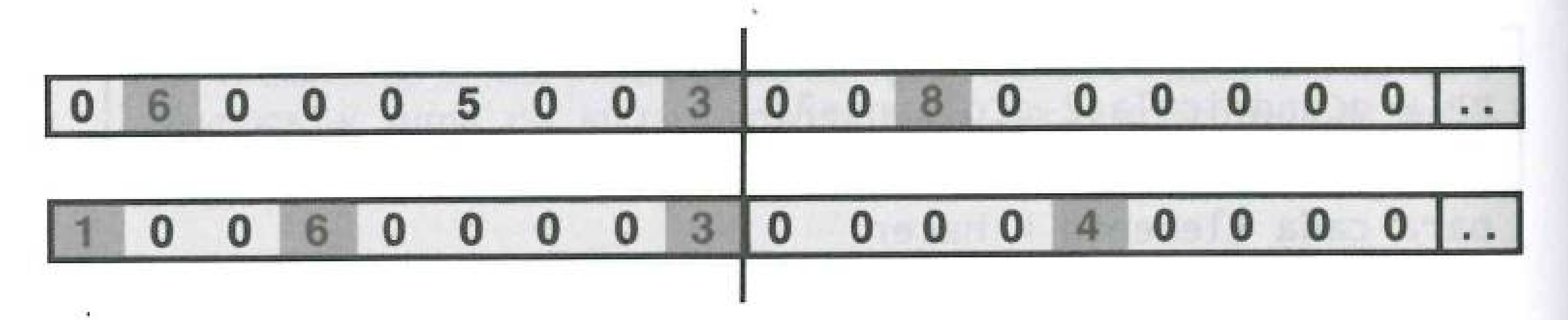
Los operadores de cruce actúan sobre parejas de individuos y producen individuos que combinan características de los progenitores. El operador crea un nuevo par de individuos combinando partes de los dos cromosomas padre. Se han estudiado diferentes métodos de cruce en una gran variedad de problemas. El operador de cruce que apliquemos al Sudoku debería respetar ciertas restricciones para no "romper" esquemas buenos producto de la evolución. Se han obtenido buenos resultados con el operador de cruce geométrico propuesto en [MOR06].

Al realizar el cruce tenemos que evitar que se modifiquen las casillas fijas que son posiciones del Sudoku preestablecidas. Basta con modificar sólo las posiciones que tienen valor 0 en el tablero original.

Si utilizamos el cruce uniforme, tenemos que añadir la restricción de que los puntos de corte tienen que respetar las filas para que se mantengan las restricciones de elementos no repetidos dentro de la fila.

Al definir un punto de cruce dentro del individuo, respetaremos los bloques correspondientes a las filas para que no se destruyan esquemas y para que se intercambien filas completas entre individuos.

En la siguiente figura se muestra un posible punto de corte (entre fila 1 y 2). Con este punto de corte y cruce uniforme (de un punto), uno de los hijos contendrá la primera fila del primer padre y el resto de las filas del segundo padre, siempre respetando las posiciones fijas que aparecen sombreadas.



Seleccionamos los posibles puntos de corte:

```
funcion calculoPuntosCruce() : vector de enteros {
  puntCruce: vector de entero;
  puntCruce[0]=8;  puntCruce[1]=17;
  puntCruce[2]=26; puntCruce[3]=35;
  puntCruce[4]=44; puntCruce[5]=53;
  puntCruce[6]=62; puntCruce[7]=71; puntCruce[8]=80;
  devolver puntCruce;
}
```

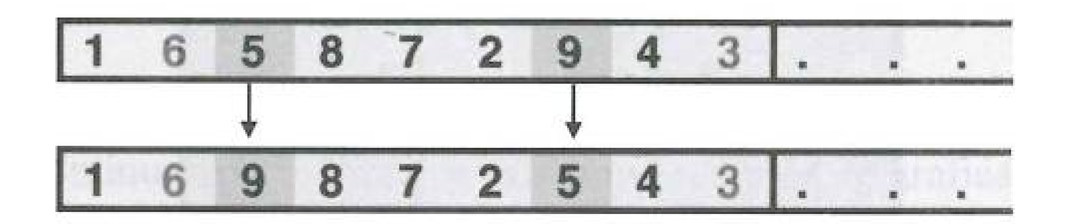
Ahora podemos aplicar el cruce uniforme con uno o varios puntos, teniendo presente que al copiar los valores en los hijos tenemos que respetar los valores fijos. Aunque con este método se han obtenido los mejores resultados, también se proponen otros métodos de cruce alternativos:

- *Cruce por emparejamiento parcial (PMX)*. Consiste en aplicar el cruce PMX a alguna de las filas del tablero que contienen las permutaciones de valores del 1 al 9.
- *Cruce por orden (OX)*. Consiste en aplicar el cruce OX a alguna de las filas del tablero que contienen las permutaciones de valores del 1 al 9.
- *Cruce alterno*. Intercala las filas de cada uno de los padres en los hijos.

2.5 Operadores de mutación

La mutación es el operador básico de alteración. Se aplica a individuos solos, realizando una pequeña modificación en alguno de sus genes o en el conjunto, permitiendo explorar nuevas zonas del espacio de búsqueda.

La mutación la vamos a aplicar sólo dentro de las filas. El esquema más utilizado es el de mutación por intercambio y es con el que mejores resultados se han obtenido: si se aplica mutación dentro de una fila, se seleccionan dos posiciones aleatorias de la fila hasta encontrar dos valores que no sean fijos y a continuación se intercambian.




```
Para cada una de las filas hacer
  Si  alea() < prob Mutación entonces
  //seleccionamos dos posiciones al azar
          repetir
            entero k1 = alea_ent(1,9);
          hasta  k1 no sea posición fija;
            repetir
            entero k2 = alea_ent(1,9);  ;
          hasta  k2 no sea posición fija;
    intercambiar posiciones k1 y k2
    mutado = cierto;
```

También se proponen las siguientes alternativas al método de mutación anterior:

- *Mutación por intercambio con la siguiente casilla.* Se intercambia la casilla con la siguiente válida dentro de la fila.

- *Rotación.* Dentro de la fila a mutar se seleccionan aleatoriamente tres elementos y se rotan hacia la derecha.

- *Regeneración.* Se inicializa aleatoriamente un gen respetando las posiciones fijas y la restricción de elementos no repetidos.

3. UN EJEMPLO DE INTERFAZ GRÁFICA

Para finalizar el análisis del proyecto se muestra un ejemplo de aplicación que incluye una interfaz gráfica que permite seleccionar diferentes parámetros para el algoritmo, así como visualizar la solución obtenida.

La aplicación permite seleccionar los parámetros del algoritmo: tamaño de la población, número máximo de generaciones, probabilidad de cruce y mutación, porcentaje de elitismo, método de selección, método de cruce y método de mutación. El tablero inicial tiene la siguiente distribución de casillas:

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

Al finalizar la ejecución del algoritmo genético podemos ver la solución obtenida, en este caso se muestra un tablero que contiene dos faltas (las casillas rodeadas por un círculo).

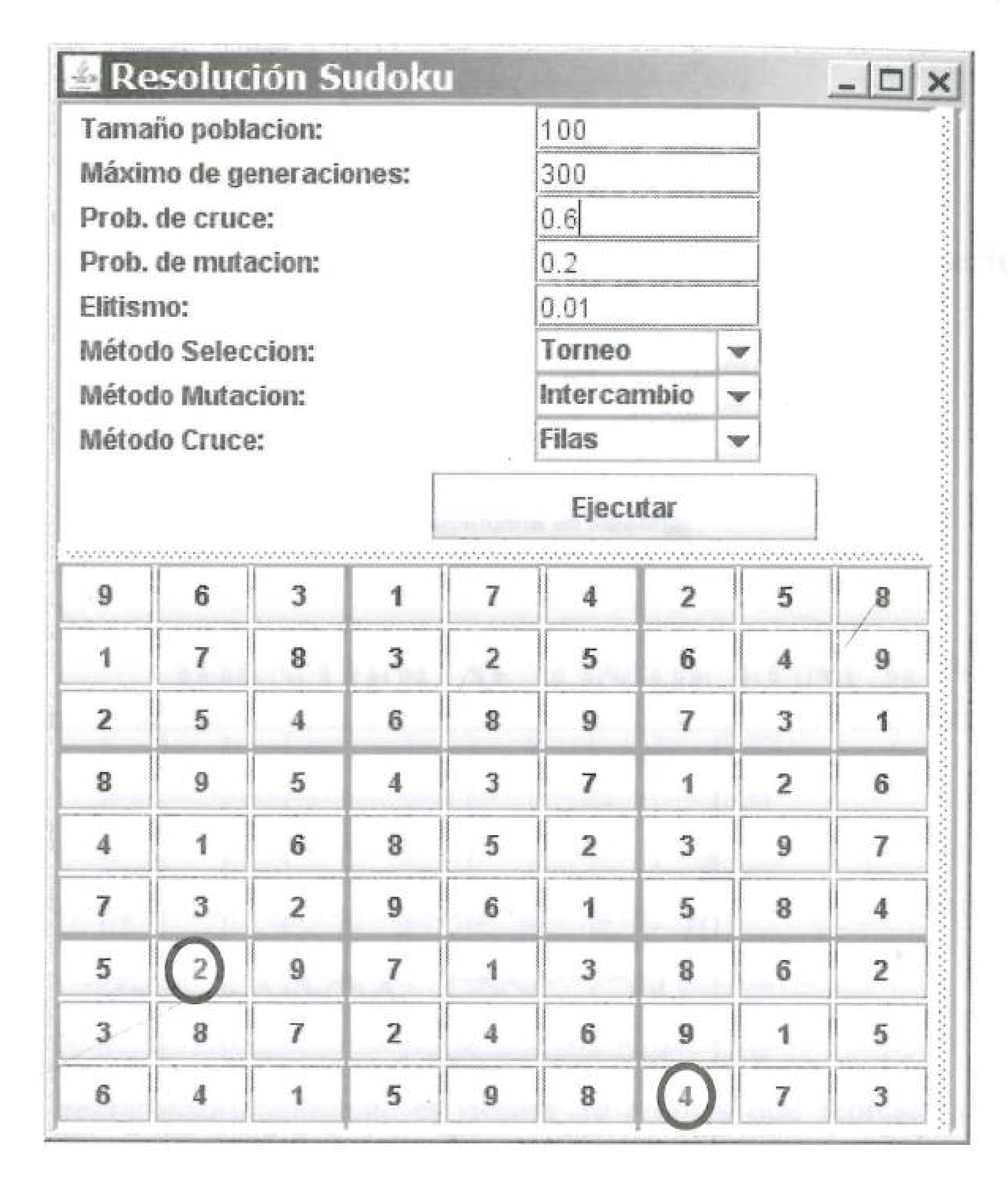
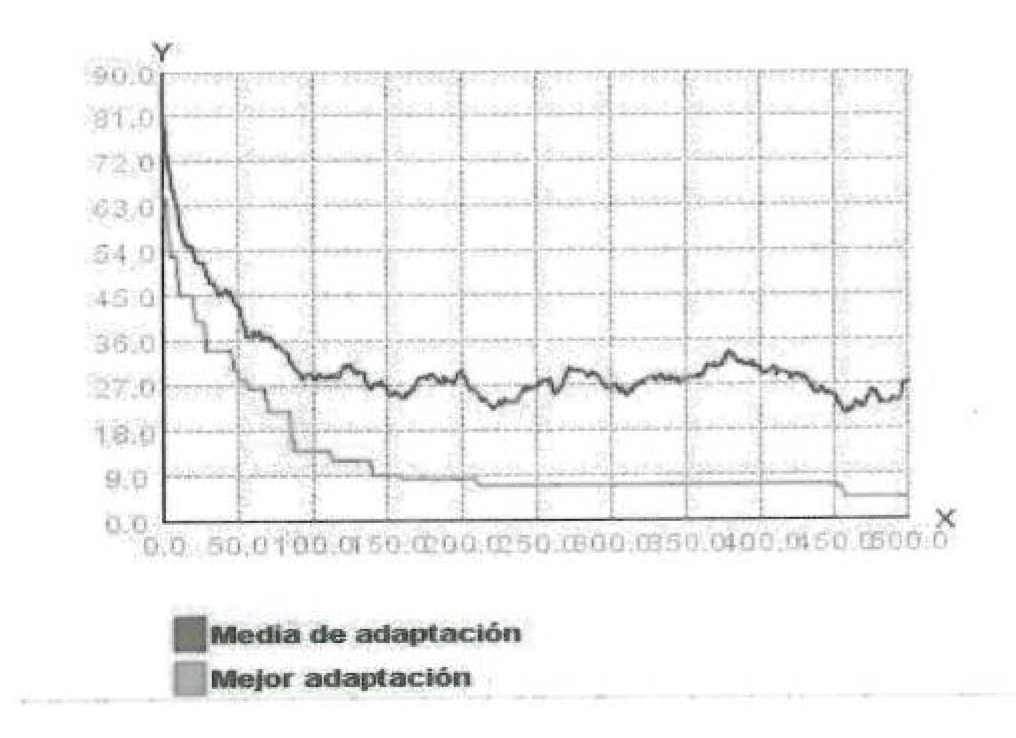
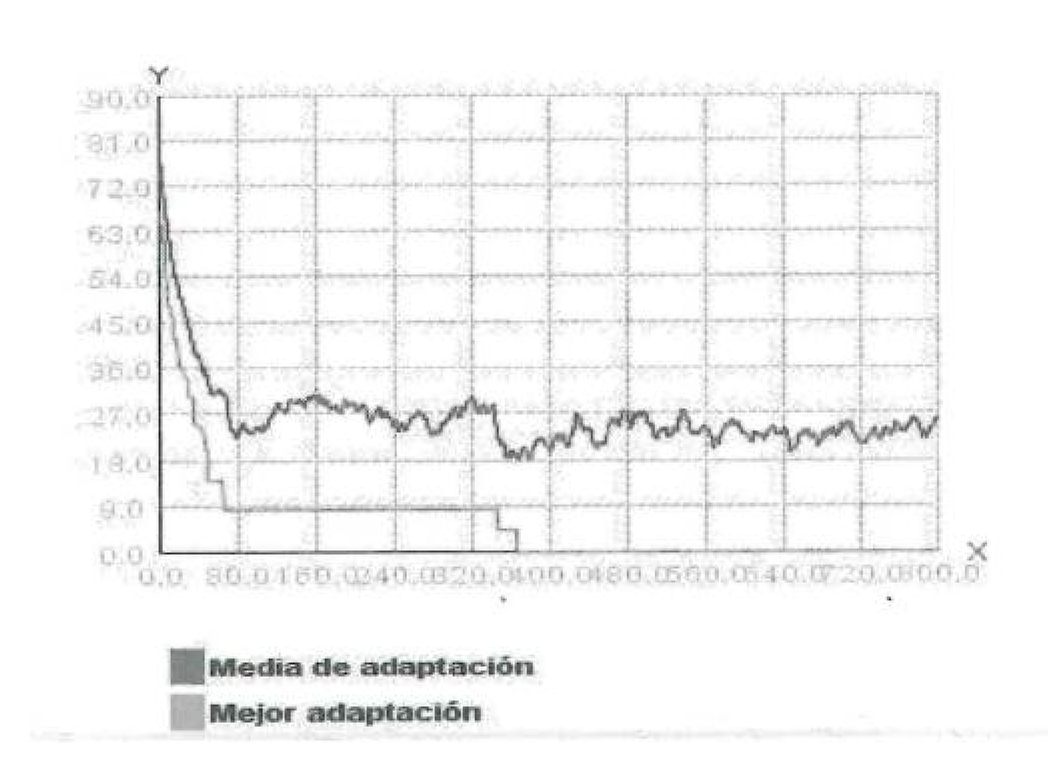


Figura 7. 3

Y a continuación se muestran algunos ejemplos de gráficas de la evolución de la mejor adaptación y del valor medio:





4. CONCLUSIONES

La mayor dificultad en este problema es encontrar una función apropiada para medir la adaptación de los individuos. El Sudoku tiene una solución única, lo que significa que no se consideran soluciones válidas los tableros con situaciones cercanas a la solución y por ello no es suficiente que la función califique tableros "medianamente" buenos.

Sí se puede afirmar que es un problema ejemplar para experimentar con diferentes parámetros, operadores genéticos sobre permutaciones y también con otros métodos de cruce como el propuesto en [MOR06], donde se estudia el cruce geométrico, que ha producido resultados realmente buenos.

Proyecto 8

IDENTIFICACIÓN DE FUNCIONES

1. DESCRIPCIÓN DEL PROBLEMA

En este proyecto usaremos la programación genética para obtener mediante técnicas de evolución la fórmula para resolver ecuaciones de segundo grado.

Tal y como se vio en el capítulo 4, en programación genética se hacen evolucionar programas codificados mediante expresiones en formato Lisp. Estos programas se representan fácilmente en forma de árboles que representan los individuos con los que trabaja el algoritmo evolutivo.

En este caso concreto lo que queremos es buscar e identificar la función o fórmula que resuelve ecuaciones del tipo:

$$ax^2 + bx + c = 0$$

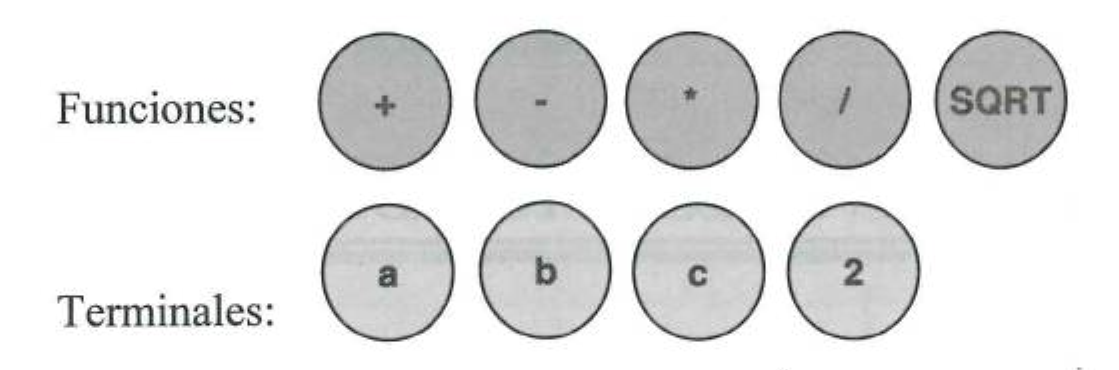
Se trata de identificar o descubrir automáticamente la función o expresión solución a partir de diferentes datos de prueba que consisten en diferentes ecuaciones de segundo grado con sus correspondientes valores de a, b y c.

Sabemos que la siguiente fórmula resuelve la ecuación anterior:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Identificación del conjunto de terminales y funciones

- El conjunto de funciones contiene 4 operadores matemáticos básicos y la función para el cálculo de la raíz cuadrada : {+, -, *, /, SQRT}.
- El conjunto de terminales contiene los valores de los coeficientes. Como constantes numéricas de podrían incluir los primeros números naturales que permiten obtener otros valores utilizando operadores. Como simplificación, utilizamos sólo la constante numérica 2, quedando un reducido grupo de símbolos terminales: {a, b, c, 2}.



Identificación de la función de adaptación

La adaptación de cada programa de la población se calcula como el resultado de su ejecución sobre un conjunto de distintas ecuaciones. La adaptación es una función del número de casos de prueba en los que produce el resultado correcto. El conjunto de ecuaciones de prueba será del tipo:

$$5x^2 - 3x + 2 = 0$$

Cada ecuación del conjunto de ejemplos contendrá un valor para las variables a, b y c. Se sustituyen dichos valores en la ecuación representada por el individuo a evaluar, y así se obtiene un valor para la x.

Se sustituye dicho valor en la ecuación de segundo grado, y se comprueba si el resultado es cero. Si es así, el individuo es una fórmula que resuelve la ecuación; si no es así, la diferencia se utilizará para el cálculo de la aptitud del individuo.

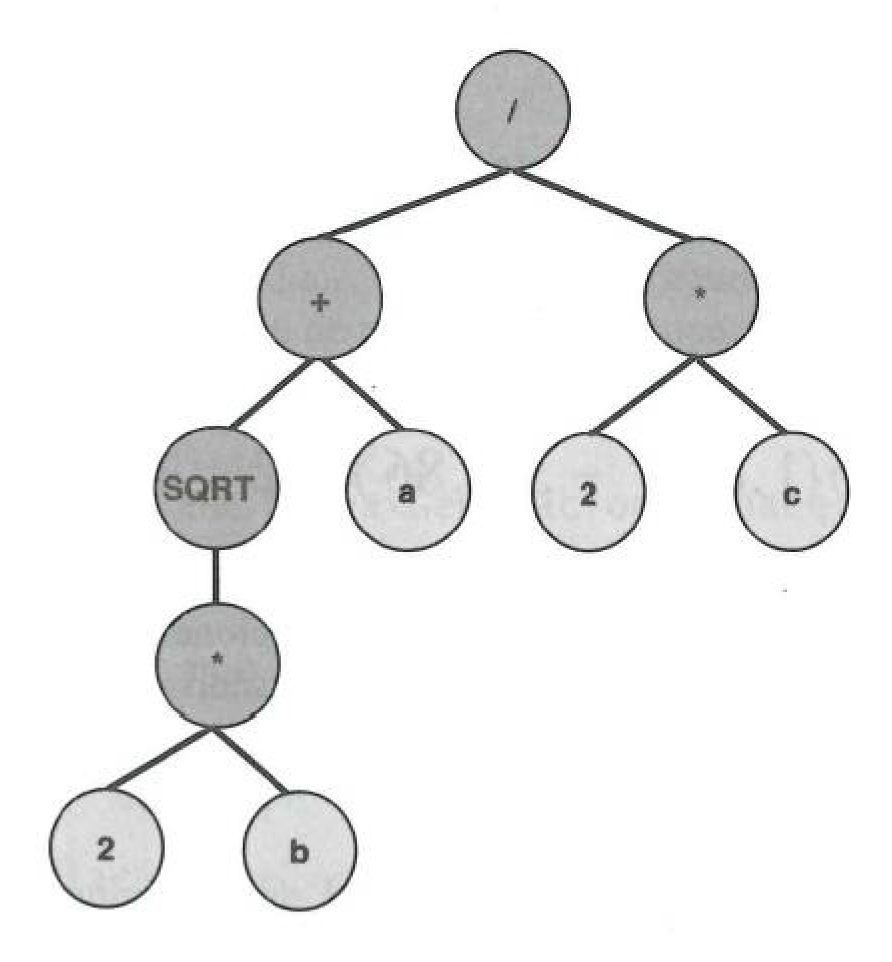
Por ejemplo, una posible solución o individuo puede ser la siguiente:

$$x = \frac{\sqrt{2b} + a}{2c}$$

que se codifica mediante la siguiente expresión :

```
(/  (+ SQRT (* 2 b)  a )(* 2 c))
```

y se puede representar mediante el siguiente árbol:



Por ejemplo, si tenemos la siguiente ecuación en el conjunto de prueba y sustituimos sus valores a, b y c en el individuo a analizar:

$$5x^2 - 3x + 2 = 0$$

obtenemos un valor para la x:

$$x = \frac{\sqrt{2b} + a}{2c}$$

$$x = \frac{\sqrt{2(3)} + 5}{2(2)}$$

$$x = \frac{\sqrt{6} + 5}{4}$$

$$x = 1.862$$

Si sustituimos ese valor en la ecuación a resolver deberíamos obtener un valor cercano a cero, que nos servirá de medida del error:

$$5x^2 - 3x + 2 = 0$$

$$5(1.86)^2 - 3(1.86) + 2 = 13.71$$

Habiendo definido el conjunto de terminales y funciones y la función de adaptación, ya podemos iniciar los pasos del algoritmo usado en programación genética. En resumen el algoritmo de programación genética para este problema tiene las siguientes características:

- Inicialmente se genera una población inicial de individuos o árboles aleatorios (expresiones) utilizando el conjunto de funciones y terminales definidos. Se pueden utilizar diferentes métodos de creación de árboles, pero los individuos generados deben ser sintácticamente correctos. Estos árboles pueden tener distintos tamaños, pero es recomendable limitar la profundidad de los mismos.

- Para evaluar la aptitud de un individuo o expresión, aplicamos la solución obtenida sobre un conjunto de ecuaciones de prueba y analizamos el número de casos de prueba en los que produce el resultado correcto.

- El programa termina cuando se ha obtenido una solución satisfactoria para un porcentaje alto de casos de prueba o se completa el número máximo de generaciones.

- Se aplican los operadores de selección, cruce y mutación.

2. DISEÑO DEL ALGORITMO

Veamos en detalle cada uno de los aspectos importantes del algoritmo. Aparte de muchos detalles, es importante definir en primer lugar el conjunto de símbolos terminales y funciones con los que va a trabajar el algoritmo:

```
cjtoTerms:vector de Cadena_caracteres[]={"a","b","c","2"};
cjtoFuns: vector de Cadena_caracteres[]={"+" , "-", "*", "/",
"sqrt"};
```

Siguiendo el esquema del algoritmos elitista, a continuación se muestra el pseudocódigo utilizado en el proyecto y posteriormente se analiza la forma de representar los individuos, el método de generación de la población inicial, la definición de la función de adaptación y los operadores de cruce y mutación.

```
poblacion.evalua();
para i=0 hasta maxGeneraciones hacer {

 //Extraemos los mejores individuos de la población

 elite = poblacion.separaMejores(tamElite);

 /* Aplicamos el proceso de selección, reproducción
    y mutación */

 poblacion.selecciona();
 poblacion.reproduce(probCruce);
 poblacion.muta(probMutacion);
 poblacion.incluye(elite);
 borrar(elite);
 poblacion.evalua();
}

devolver MejorIndividuo;
```

2.1 Representación de los individuos

Los individuos contendrán los atributos comunes del cromosoma visto en otras aplicaciones: adaptación, puntuación, puntuación acumulada, etc. Además ahora necesitamos la estructura de árbol para la representación sintáctica de la expresión o programa:

```
tipo TIndividuo =  registro{
   TArbol arbol;      // expresión o fórmula
   real adaptación; // función de evaluación
   real puntuacion; // punt.relativa:adaptación/sumadaptación
   real punt_acu;    // puntuación acumulada
   booleano elite;   // elitismo
}
```

Y la población es una colección de individuos:

```
tipo Tpoblacion: vector de Tindividuo;
```

Los individuos o árboles están formados por la combinación de nodos correspondientes a las funciones u operadores (`+, -, *, /, sqrt`) y de los nodos terminales correspondientes a los operandos (`a, b, c, 2`). Estos árboles codifican fórmulas que representan soluciones a una ecuación de segundo grado genérica.

Los árboles pueden contener operadores con uno y dos operandos, por lo que podemos considerar dos nodos hijos. En la representación de funciones de dos argumentos (operandos), los hijos izquierdo y derecho hacen referencia al primer y segundo operando respectivamente. Para las funciones con un operando sólo se utiliza el hijo izquierdo y el hijo derecho (Hd) estará a nulo.

Por utilidad también almacenamos el número de nodos del árbol y la profundidad. Para ello utilizamos la siguiente representación:

```
tipo TArbol =  registro{
  cadena dato;           // operando u operador
  TArbol Hi;             // hijo izquierdo
  TArbol Hd;             // hijo derecho
  TArbol padre           // padre
  entero num_nodos;      // número de nodos
  entero profundidad;   // profundidad del árbol
  booleano esRaiz;       // es nodo raíz
  booleano esHoja;       // es nodo terminal
  . . .
}
```

2.2 Generación de la población inicial

Se han implementado los dos tipos de inicializaciones vistos en el capítulo 4 (completa y creciente) para crear los árboles en función de la profundidad máxima especificada como parámetro. En el primer método se generan todas las

ramas del árbol hasta la profundidad máxima con nodos que contienen como expresión una función y sólo en el último nivel (hojas) se sitúan los terminales.

En la otra alternativa se pueden incluir terminales aunque no se haya alcanzado la profundidad máxima. Las restricciones que se van a aplicar al crear los individuos son las siguientes:

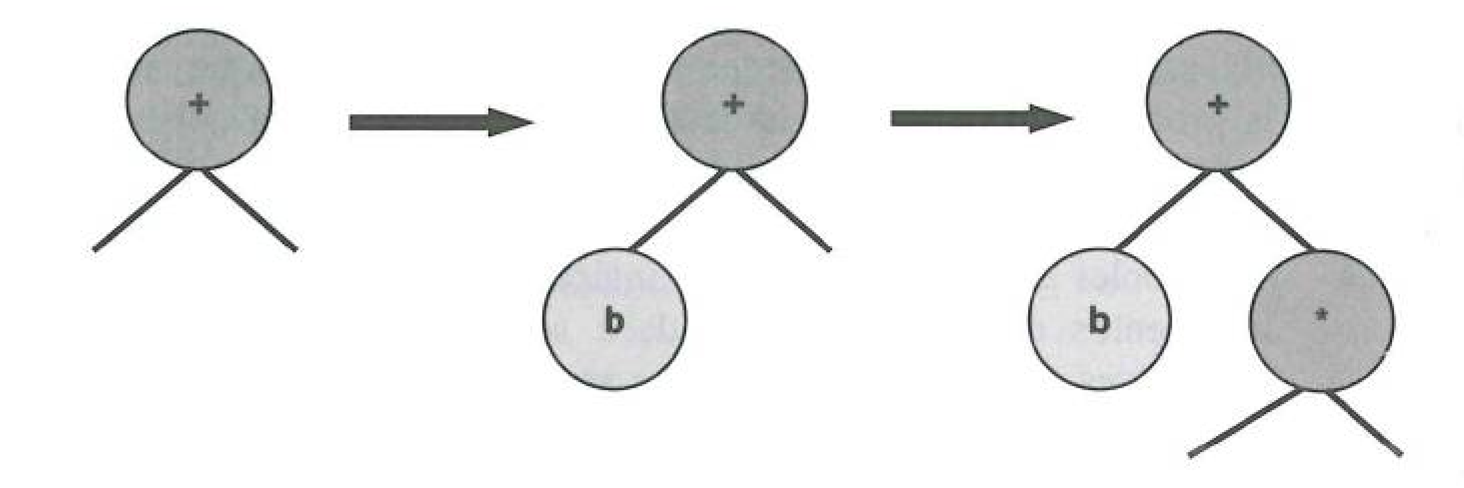
- Los árboles generados deben ser sintácticamente correctos: el número de argumentos de los operadores debe ser el correcto y los operandos (constantes o variables) sólo pueden ser hojas del árbol.
- La profundidad de los árboles se limitará mediante un parámetro configurable mediante la interfaz de usuario. Esto permite evitar estructuras demasiado grandes que implican un proceso con alto coste a nivel computacional. El operador de cruce tiende a hacer crecer los árboles y puede hacer que el tiempo de evaluación sea excesivo. Por ello, se buscará el valor adecuado de profundidad para que el programa represente una buena estrategia y no afecte demasido al rendimiento del algoritmo.

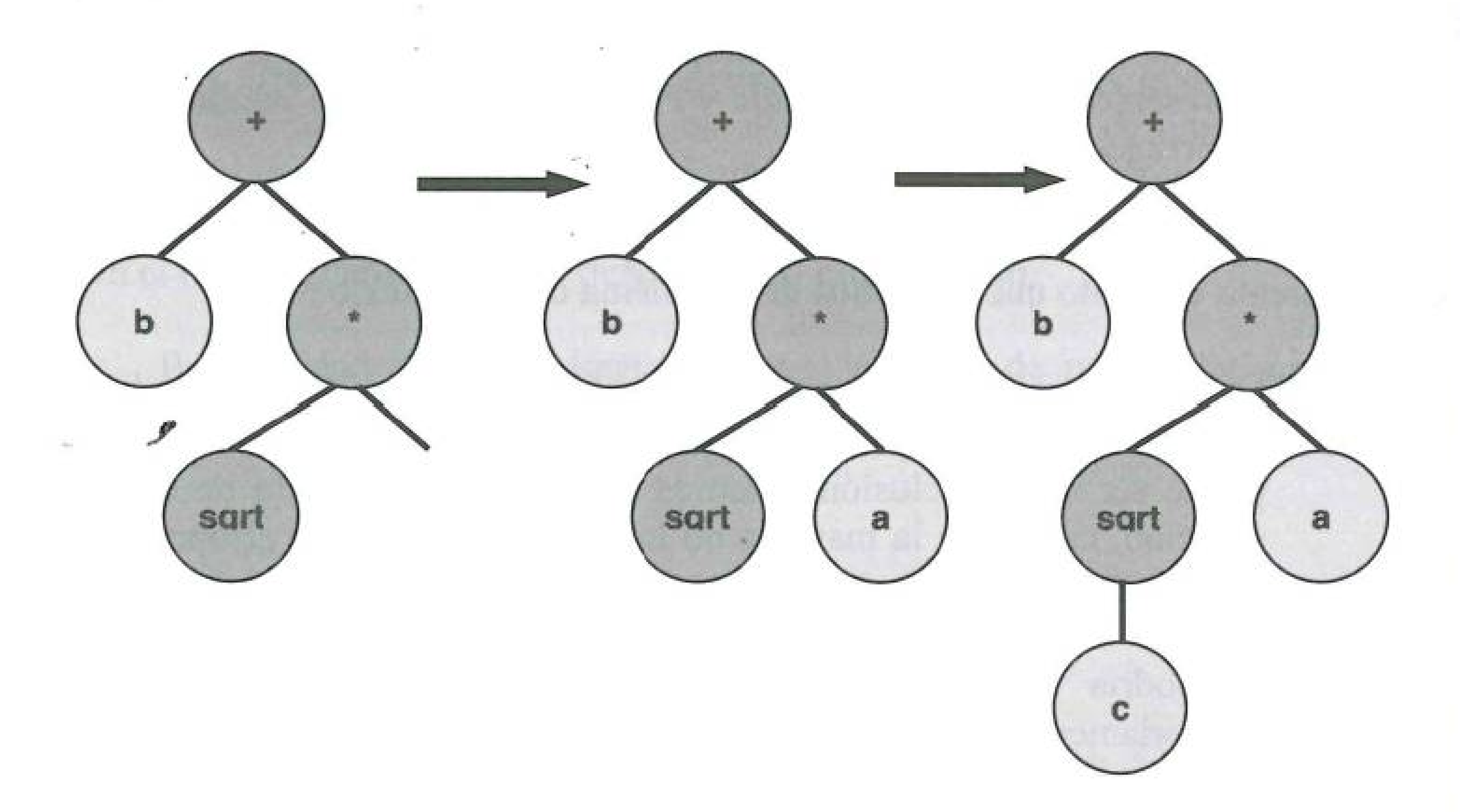
Y opcionalmente se pueden imponer otras restricciones, dependiendo del conocimiento explícito que tengamos del problema o del contexto.

- Que el primer nodo de cada árbol sea "/", sin incluir ningún nodo interno con dicho operador para partir de una población inicial "buena". Esto puede ser una conclusión obtenida mediante un conjunto de ejemplos solución, en la que la mayoría de los ejemplos incluyen el operador de división "/".
- Se podría imponer que cuando obtenemos un nodo "-", elegimos aleatoriamente si tiene dos hijos (operador de resta) o si se corresponde con el cambio de signo, al que le corresponde sólo un hijo.

Una vez creado el individuo puede surgir el problema de que la evaluación del individuo sea imposible o indeterminada, pues tratamos con operadores matemáticos que pueden dar lugar a operaciones no definidas (raíz cuadrada de un número negativo o división por 0). Esto se podría solucionar desechando los individuos incorrectos o asignarle un valor muy bajo de adaptación, al ser un problema de minimización.

Como se muestra en el siguiente gráfico los árboles se crean mediante inicialización creciente (en este caso de profundidad 3):





A continuación se muestra el pseudocódigo de una función que genera un árbol con una posible función-solución manteniendo las restricciones anteriores y evitando que los árboles tengan una sola hoja:

```
funcion construir_arbol(TArbol padreP, entero profMax,
                        booleano esRaizP, esHiP) {

 real valor;
 entero nuevaProf = profundidad + 1;

    /* Comprobamos si va a ser hoja o nodo interno */

    valor = aleatorio(0,1) * 2;
    padre = padreP;
    esHi = esHiP;
    esRaiz = esRaizP;
    numNodos = 1;
    si ((valor = 0) or (profundidad + 1 = profMax))
    {
      /* seleccionamos un terminal : hoja */
      valor = aleatorio(cjtoTerms.longitud);
      si (valor = 5) valor = 4;
      valor = cjtoTerms[valor];
      esHoja = cierto;
    }
    eoc
    {
      real hojaRandom = aleatorio(0,1);
      si hojaRandom <= 0.8) {


  /* Creamos sus hijos */
       valor = aleatorio(cjtoTerms.longitud);
       si (valor = 5) valor = 4;
       valor = cjtoFuns[valor];
       hi = construir_arbol(this,nuevaP,false,true);
       numNodos = numNodos + hi.getNumNodos();

       /*Si la función tiene un solo argumento el árbol
         solo tendrá hijo izquierdo*/

       si (cjtoFuns[(int)rnd] == ("sqrt"))
                                hd = null;
        eoc
         {
          hd = construir_arbol(this,nuevaP,false,false);
          numNodos = numNodos + hd.getNumNodos();
         }
        esHoja = false;
       }
       eoc {

     /* Generamos un terminal a profundidad
        menor de altura máxima */

        valor = aleatorio(cjtoTerms.longitud);
        si (valor = 5) valor = 4;
```

```
            valor = cjtoTerms[valor];
            esHoja = true;
            hi = null;
            hd = null;
          }
        }
    }
```

Según esta función, cuando se alcanza la profundidad máxima, se genera una hoja correspondiente a un operando. Si no se ha alcanzado la profundidad máxima, se decide aleatoriamente entre un operador y un operando y se genera el árbol que corresponda.

2.3 Función de adaptación

La adaptación o aptitud de un individuo es el margen de error obtenido al aplicar la fórmula que codifica un individuo sobre un conjunto de ecuaciones de prueba. Para ello disponemos de un archivo de texto que contiene una batería de ejemplos de posibles ecuaciones. La primera columna contiene los valores de "a", la segunda de "b" y la tercera de "c".

```
3 -6 -10
7 -3 9
-4 4 7
5 2 -1
-6 6 0
. . .
```

Podemos almacenar los datos en una matriz que luego será útil a la hora de evaluar:

```
ecuaciones : matriz de entero {
                {3, -6, -10},
                {7, -3, 9},
                {-4, 4, 7},
                {5, 2, -1},
                {-6, 6, 0},
                  . . .
             }
```

Para cada conjunto de prueba se realizan los siguientes pasos:

- Sustituimos los valores de a, b y c en la expresión.
- Obtenemos el resultado de evaluar la expresión para los valores correspondientes de a, b y c.

- Sustituimos el resultado de la evaluación en la ecuación de segundo grado, calculamos el resultado y se lo sumamos a la aptitud.

Para evitar tener que aplicar todas las restricciones y simplificar el problema todas las ecuaciones de prueba que se utilizan tienen raíces reales (no complejas) y el coeficiente del término de segundo grado es distinto de 0 para no obtener divisiones por 0 en la ecuación solución.

A continuación se muestra un esquema de función que calcula y devuelve la aptitud de un individuo de la población:

```
funcion evalua(ecuaciones:matriz de entero) {

/* Resultado de evaluar el árbol con los coeficientes
   de las ecuaciones que recibe como parámetro */

  real resul,res;
  real acumulado = 0;
  para i=0 hasta ecuaciones.longitud {
   resul=arbol.evalua(ecuacion[i][0],ecuacion[i][1],
                      ecuacion[i][2]);

   res=evalSegundoGrado(ecuacion[i][0],ecuacion[i][1],
                        ecuacion[i][2], resul));

   acumulado +=100*abs(res);
   }
  devolver (acumulado/ecuaciones.longitud);
}
```

La función `evaluaSegundoGrado` se encarga de evaluar la ecuación de segundo grado de forma $ax^2 + bx + c = 0$ conociendo los coeficientes y el valor de la x:

```
funcion evalSegundoGrado(real a, b, c, x) {
     devolver ((a * x * x) + (b * x) + c);
 }
```

La parte principal de la evaluación la realiza la función `evalua`, que aplicada al árbol devuelve el valor de evaluar la expresión del árbol con `a`, `b` y `c`.

```
funcion evalua(real a, b, c) {
 real resi, resd, res = 0;
```

```
    si (esHoja) entonces
      si (valor == "a") entonces devolver a;
       eoc
        si (valor == "b") entonces devolver b;
         eoc
          si (valor == "c") entonces devolver c;
           eoc
            si (valor == "2")
                        entonces devolver 2;
            eoc devolver 0;
     eoc {
           resi = hi.evalua(a, b, c);
           si (valor == "sqrt") entonces {
                       si (resi < 0) entonces
                             res = -MAX_VALUE;
                       eoc
                             res = sqrt(resi);
                 }
           eoc {
                  resd = hd.evaluar(a, b, c);
                  si (valor  == "*") entonces
                             res = resi * resd;
                  eoc
                   si (valor  == "/") entonces {
                     si (resd != 0) entonces
                             res = resi/resd;
                     eoc
                      res = MAX_VALUE;
                    }
                   eoc
                     si (valor  == "+") entonces
                             res = resi + resd;
                     eoc
                       si (valor  == "-") entonces
                        res = resi - resd;
               }
          }
      devolver res;
    }
}
```

2.4 El operador de cruce

En este proyecto se utiliza el cruce más común: cruce por intercambio de subárboles, en el que se selecciona un nodo aleatorio de cada padre y se intercambian los subárboles que hay bajo estos nodos. Para elegir el nodo de corte podemos utilizar diferentes métodos, por ejemplo el método que utiliza diferentes probabilidades de selección según el tipo de nodo.

El algoritmo básico de intercambio de subárboles se muestra en el siguiente pseudocódigo:

```
TArbol subarbol1, subarbol2;
entero num_nodos;

num_nodos=minimo(num_nodos(padre1.arbol),
                 num_nodos(padre2.arbol));
nodo_cruce = alea_entero(1,num_nodos);
hijo1.arbol = padre1.arbol;
hijo2.arbol = padre2.arbol;
subarbol1 = hijo1.arbol.BuscarNodo(nodo_cruce);
subarbol2 = hijo2.arbol.BuscarNodo(nodo_cruce);
hijo1.arbol.SustituirSubarbol(nodo_cruce, subarbol2);
hijo2.arbol.SustituirSubarbol(nodo_cruce, subarbol1);
hijo1.adaptacion = adaptacion(hijo1);
hijo2.adaptacion = adaptacion(hijo2);
```

La función `BuscarNodo(n)` devuelve el subárbol bajo el nodo que ocupa la posición `n` del árbol. La función `SustituirSubarbol(n,a)` sustituye el subárbol que se encuentra bajo el nodo que ocupa la posición `n` por el subárbol `a`.

Para evitar que el operador de cruce genere árboles excesivamente grandes podemos podar el árbol cuando se supere una altura máxima permitida o utilizar cualquier técnica que limite la profundidad de los árboles, limitando el número de nodos o la longitud máxima de las expresiones.

2.5 Operador de mutación

Para la mutación se puede utilizar cualquiera de los métodos vistos en el capítulo 4: mutación funcional, mutación terminal, mutación de árbol o mutación de permutación.

Un posible operador de mutación que abarca los dos primeros métodos es el método que muta un nodo del árbol elegido al azar: si el nodo es un terminal cambiamos su valor por otro terminal, y si es una función cambiamos su valor por el de otra función.

Si el nodo corresponde a una función, debemos sustituir de manera aleatoria la función existente en ese nodo por otra, teniendo en cuenta el número de operandos de la función original. Por ejemplo, si la función era + se cambia por /.

```
booleano mutado;
entero i, nodo_mut;
real prob;
```

```
TArbol subarbol;
para cada i  desde 0  hasta tam_pob  hacer{
    mutado =  falso;
    // se genera un número aleatorio en [0 1)
    prob = aleatorio(0,1);
    si (prob < prob_mut) entonces {
      mutado =  cierto;
      // se selecciona un nodo
      nodo_mut = alea_entero(1,num_nodos(pob[i].arbol));
      subarbol = pob[i].arbol.BuscarNodo(nodo_mut);
      si no EsHoja(subarbol)
          CambiarOperador(subarbol)
      eoc
          CambiarOperando(subarbol)
    }
   si (mutado)
     pob[i].adaptacion = adaptacion(pob[i], lcrom); . . .
```

En caso de que se produzca mutación la función elige uno de los nodos del árbol aleatoriamente y lo cambia según sea operador u operando. La función `CambiarOperando` cambia el operando por otro distinto elegido aleatoriamente. La función `CambiarOperador` cambia el operador por otro distinto elegido aleatoriamente, respetando el número de argumentos requerido por cada operador, lo que puede implicar la modificación del árbol.

Un método de mutación muy común es el de mutación de árbol, en el que se selecciona un nodo al azar y generamos de forma aleatoria un nuevo subárbol por debajo de ese nodo. Eso implica cambiar parte del pseudocódigo anterior:

```
. . .
subarbol = pob[i].arbol.BuscarNodo(nodo_mut);
si no EsHoja(subarbol)
     construirArbol(subarbol,nuevaprofundidad)
eoc
. . . .
```

Como veremos más adelante, en la interfaz de usuario de la aplicación se ofrece la posibilidad de seleccionar otros métodos de mutación implementados:

- Mutación de Terminal: seleccionamos al azar un nodo terminal (una hoja) y sustituimos de manera aleatoria el terminal en ese nodo por otro.

- Mutación Funcional: seleccionamos al azar un nodo que no sea una hoja del árbol, y sustituimos de manera aleatoria la función existente en ese nodo por otra con la misma aridad.

Se ha introducido un mecanismo de elitismo para que el algoritmo alcance el resultado con mayor rapidez. Antes de realizar la selección se extrae un

porcentaje de individuos con mejor aptitud. Después de la selección, reproducción y mutación, se vuelven a introducir los cromososmas de la élite en la población.

3. UN EJEMPLO DE INTERFAZ GRÁFICA

Para finalizar el análisis del proyecto se muestra un ejemplo de aplicación que incluye una interfaz gráfica que permite seleccionar diferentes parámetros para el algoritmo, así como visualizar la solución obtenida. En la siguiente figura se muestra la interfaz gráfica de la aplicación.

Como se puede observar en la figura, la aplicación permite seleccionar los parámetros del algoritmo: tamaño de la población, número de generaciones, probabilidad de cruce y mutación y tipo de mutación (de árbol, terminal y funcional). Al finalizar la ejecución del algoritmo genético podemos observar la gráfica de evolución.

Poyecto Identificación de funciones

Parámetros

Tamaño de población 100 Número generaciones 600
Prob. de cruce 0.6 Prob. de mutación 0.1
Elite 5 Profundidad máxima 5
Mutación Terminal Mutación Funcional

Aceptar Salir

Resultados

```
*(a,/(+(sqrt(2),2),*(+(2,2),*(a,2)))) Aptitud: 7.17399787902832

>> Generación 568
 *(a,/(+(sqrt(2),2),*(+(2,2),*(a,2)))) Aptitud: 7.17399787902832

>> Generación 569
 *(a,/(+(sqrt(2),2),*(+(2,2),*(a,2)))) Aptitud: 7.17399787902832

>> Generación 570
 *(a,/(+(sqrt(2),2),*(+(2,2),*(a,2)))) Aptitud: 7.17399787902832

>> Generación 571
 *(a,/(+(sqrt(2),2),*(+(2,2),*(a,2)))) Aptitud: 7.17399787902832

>> Generación 572
 *(a,/(+(sqrt(2),2),*(+(2,2),*(a,2)))) Aptitud: 7.17399787902832

>> Generación 573
 *(a,/(+(sqrt(2),2),*(+(2,2),*(a,2)))) Aptitud: 7.17399787902832
```

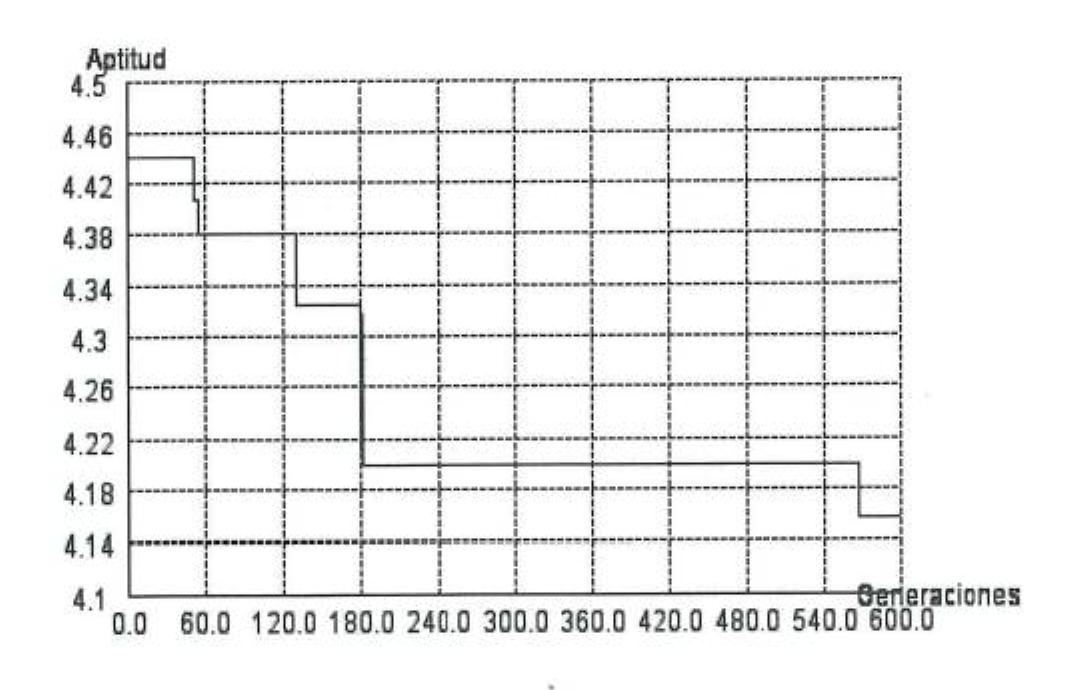
También se va mostrando por consola todo el proceso evolutivo:

```
. . .

>> Generacion 269
Ecuación: /(c,+(sqrt(*(c,c)),+(-(0,0),2)))
Aptitud: 3.7492151260375977

. . .
. . .
```

Al finalizar la ejecución podemos ver la gráfica de evolución:



4. CONCLUSIONES

Después de muchas pruebas realizadas con diferentes parámetros llegamos a la conclusión de que tiene mucha importancia la selección de las ecuaciones de prueba, que han de seleccionarse de forma que sirvan para diferenciar correctamente los árboles con fórmulas de alta adaptación de los que codifican fórmulas con muy baja adaptación.

Se han obtenido soluciones cercanas pero nunca han llegado a converger al óptimo. Al variar los parámetros, se ha visto la importancia de los operadores de mutación y regeneración para evitar el estancamiento de la población en óptimos locales, al crear individuos nuevos que aportan los cambios necesarios.

El hecho de que los individuos sean expresiones matemáticas que pueden originar tanto divisiones por cero como raíces de números negativos implica que se generan elementos que podrían ser válidos para generaciones posteriores pero que quedan descartados por no obtener buenos valores de adaptación.

Proyecto 9

GENERACIÓN DE ESTRATEGIAS DE RASTREO

1. DESCRIPCIÓN DEL PROBLEMA

En este proyecto usaremos la programación genética para obtener mediante técnicas de evolución estrategias de rastreo para simular los movimientos de una hormiga artificial que busca comida en un determinado mapa, según el problema planteado por Robert J. Collins y David R. Jefferson y luego aplicado por John Koza (KOZ90).

Tal y como se vio en el capítulo 4, en programación genética se hacen evolucionar programas. Estos programas se representan fácilmente en forma de árboles que representan los individuos con los que trabaja el algoritmo evolutivo.

Este proyecto consiste en buscar una estrategia de movimiento que tiene que seguir una hormiga artificial para encontrar toda la comida situada a lo largo de un rastro irregular. El objetivo es utilizar programación genética para obtener el programa que realiza esta tarea (KOZ90). La hormiga artificial tiene que simular un recorrido sobre un tablero de 32 x 32 que representa una estructura toroidal. La hormiga comienza en la esquina superior izquierda del tablero, identificada por las coordenadas (0,0), y mirando en dirección Este.

Un modelo de rastro propuesto, con el que se han realizado diversos estudios, se conoce como "rastro de Santa Fe", y tiene una forma irregular compuesta de 89 "bocados" de comida. El rastro presenta huecos de una o dos posiciones, que también pueden darse en los ángulos.

En resumen lo que queremos es resolver el siguiente problema: dado un tablero en el que existe un camino discontinuo de casillas con comida, y una hormiga que es capaz de avanzar, girar y comprobar si tiene comida delante, encontrar un programa Lisp que, ejecutado en un bucle infinito pero limitando el número de avances y giros a ejecutar, haga recorrer a la hormiga a través de la mayor cantidad posible de posiciones marcadas del tablero. La posición inicial de la hormiga es la esquina superior izquierda y su orientación es hacia la derecha. Sin olvidar el esquema evolutivo básico, en este proyecto modificaremos algunos elementos para adaptarlos al enfoque de programación genética.

Ahora los individuos se representan mediante árboles. Los nodos intermedios de los árboles pueden ser cualquiera de estas funciones: PROGN3, PROGN2 y SIC. Los nodos hoja pueden ser: IZQUIERDA, DERECHA Y AVANZA. Cada árbol representa un recorrido y hay que ejecutarlo tantas veces como pasos máximos hayamos puesto como límite (400). Cada vez que se llega a una hoja, se cuenta un paso. La aptitud para este problema es la cantidad de bocados de comida conseguidos al realizar el recorrido asociado a la estrategia que hay codificada en un individuo.

En la siguiente figura podemos ver una representación del tablero y la disposición de la comida según el rastro de Santa fe.

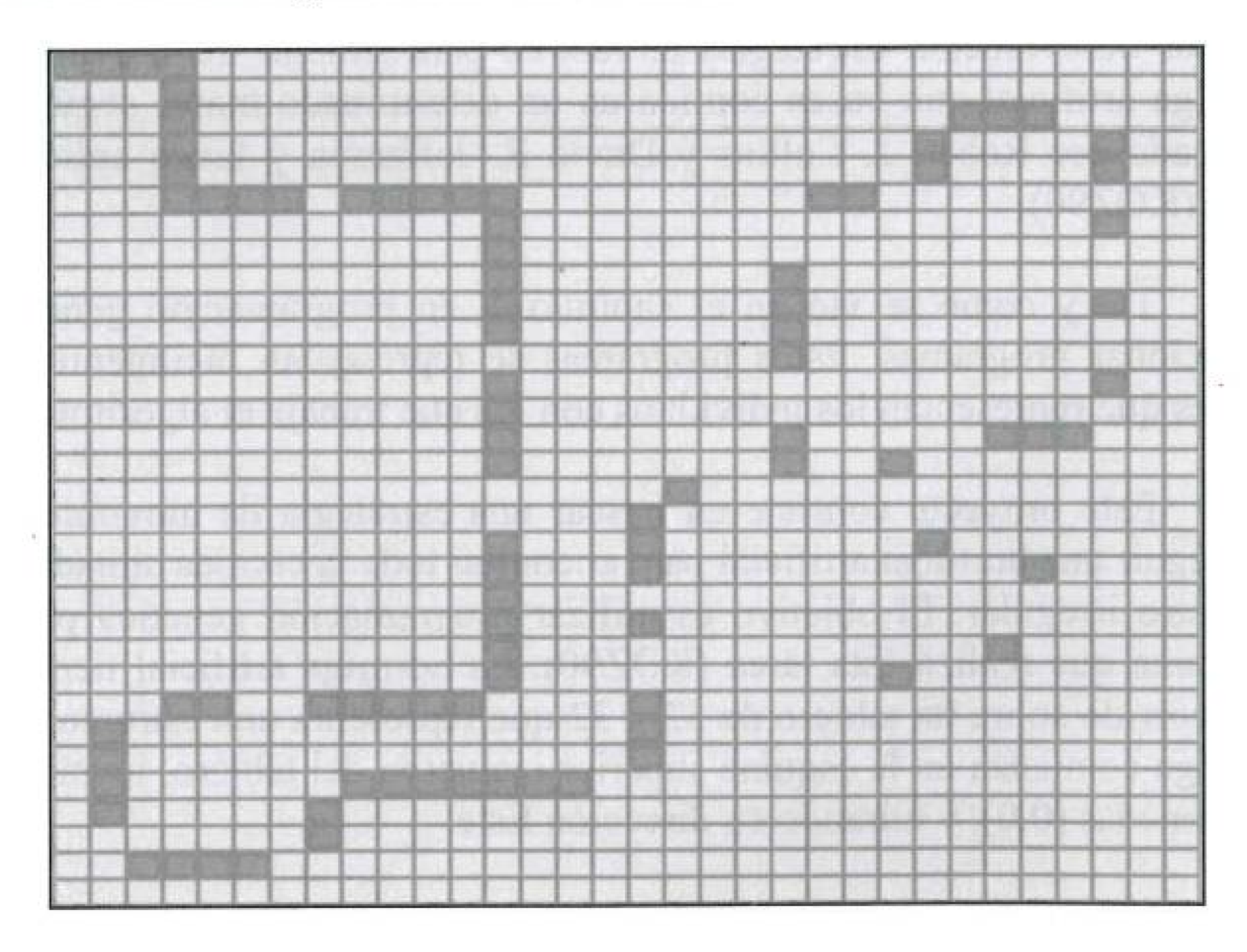


Figura 1

En Lisp los programas, aparte de la asignación de valores a variables y la salida de resultados, son una función de un conjunto de operandos y operadores. El conjunto de operandos (terminales) para este problema es:

```
T={AVANZA,IZQUIERDA,DERECHA}
```

AVANZA mueve la hormiga hacia delante en la dirección a la que mira en ese momento, DERECHA gira la hormiga 90° a la derecha e IZQUIERDA la gira 90° a la izquierda. La ejecución de nodos IZQUIERDA, DERECHA y AVANZA durante la evaluación de la aptitud hace que el juego contabilice un paso o movimiento.

Un posible conjunto de funciones u operadores para este problema es el siguiente:

```
O = {SIC,PROGN2,PROGN3}
```

La función SIC toma dos argumentos. Por ejemplo `(SIC arg1 arg2)` se define como

```
si hay comida delante entonces
          ejecuta arg1
en otro caso
          ejecuta arg2
```

`PROGN2` y `PROGN3` son instrucciones basadas en conectivas de LISP que fuerzan la ejecución de sus argumentos en un determinado orden. Las conectivas PROGN2 y PROGN3 toman dos y tres argumentos respectivamente. Con estos conjuntos de generan individuos como el siguiente:

```
(SIC (PROG3N  (AVANZA IZQUIERDA AVANZA))
     (PROG2N  (IZQUIERDA AVANZA)))
```

Las expresiones de este tipo se representan fácilmente mediante árboles formados por funciones (operadores) y terminales (operandos).

En el capítulo 4 se explica detalladamente este tipo de representación. En la figura 2 se muestra un ejemplo de programa representado en forma de árbol.

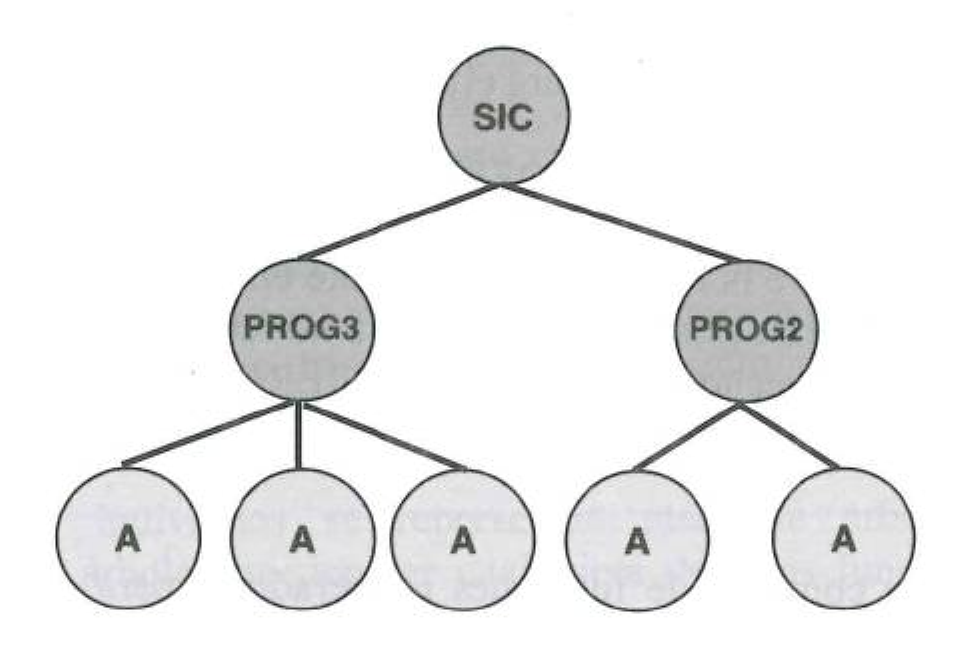


Figura 2

El programa representado por un individuo se ejecuta para calcular el número de trozos de comida que obtiene la hormiga al seguir la estrategia codificada en dicho programa.

Para limitar el proceso se fija un tiempo máximo para aplicar dicha estrategia. A continuación se muestra un ejemplo de estrategia obtenida por un programa de aptitud de valor 72 y su programa correspondiente:

```
(PROGN3 (SIC(AVANZA IZQUIERDA) PROGN2(AVANZA IZQUIERDA)
      SIC(SIC(AVANZA DERECHA) SIC(SIC(AVANZA IZQUIERDA)
      PROGN2 (DERECHA DERECHA))))
```

Si ejecutamos repetidamente la estrategia codificada en el problema anterior podremos ver el recorrido que sigue la hormiga reflejado en la figura 3. Los cuadros que aparecen en color rojo son los bocados conseguidos por la hormiga.

En esta estrategia concreta podemos ver cómo la hormiga mantiene una ruta sin salirse del camino, aunque como puede verse no consigue comer todos los trozos colocados en el rastro de Santa Fe.

El recorrido realizado por la hormiga es el siguiente:

Figura 3

El algoritmo de programación genética para este problema tiene las siguientes características:

- Inicialmente se genera una población inicial de individuos o árboles aleatorios (programas) utilizando el conjunto de funciones y terminales existentes. Se pueden utilizar diferentes métodos de creación de árboles, pero los individuos generados deben ser sintácticamente correctos. Estos árboles pueden tener distintos tamaños, pero es recomendable limitar la profundidad de los mismos.

- Para evaluar la aptitud de un individuo, el código del programa se ejecuta sobre un mapa que contiene comida según el modelo del rastro de Santa Fe. Ejecutando el programa sobre dicho tablero se obtiene un recorrido que se puede "calificar" según el número de trozos de comida que ha conseguido la hormiga. Un factor importante es el tiempo que se tarda en comer todos los trozos de comida.

- Consideramos que cada operación de movimiento (AVANZA) o giro (IZQUIERDA, DERECHA) consume una unidad de tiempo. En nuestra

versión del problema limitaremos el tiempo a 400 pasos. El tablero se va actualizando a medida que desaparece la comida.

- El programa termina cuando la hormiga consigue todos los bocados de comida o finaliza el tiempo disponible.
- Se aplican los operadores de selección, cruce y mutación.

2. DISEÑO DEL ALGORITMO

En primer lugar vamos a analizar la representación del terreno sobre el que se encuentra la comida. Utilizamos una estructura toroidal que representaremos mediante una matriz con valores 0 (no hay comida) y 1 (sí hay comida).

Para simular la estructura toroidal suponemos que cuando la hormiga llega a uno de los extremos de la matriz, continúa su recorrido por el lado opuesto. La matriz es parecida a la siguiente:

```
matriz: matriz de entero={
                   {1,1,1,1,0,0,0,0,0,. . . }
                   {0,0,0,1,0,0,0,0,0,. . . }
                   {. . .                   }
        }
```

También utilizaremos los siguientes datos:

```
tipo TMapa = registro{
  Ttablero tab;//matriz con comida;
  entero bocados;//contador de bocados comidos
  entero pasos; //contador de pasos realizados
  enumerado Tdireccion {ESTE, OESTE, NORTE, SUR};
  Tdireccion direccion; //dirección que lleva la hormiga
}
```

2.1 Representación de los individuos

Los individuos contendrán los atributos comunes a los individuos de otras aplicaciones: adaptación, puntuación, puntuación acumulada, etc. Además ahora necesitamos la estructura de árbol para la representación sintáctica de una expresión o programa:

```
tipo TIndividuo =  registro{
 TArbol arbol;// estrategia de rastreo
 real adaptación;// función de evaluación
 real puntuacion;//puntuación relativa:adaptación/sumadaptación
 real punt_acu;// puntuación acumulada
 booleano elite;// elitismo
}
```

Los individuos o árboles están formados por la combinación de nodos correspondientes a funciones (SIC, PROGN2, PROGN3) y de nodos terminales correspondientes a los operandos (IZQUIERDA, DERECHA, AVANZA). Estos árboles codifican y representan la estrategia de rastreo que tiene que seguir la hormiga para conseguir la máxima cantidad de comida.

Los árboles pueden contener operadores con dos y tres operandos, por lo que podemos considerar tres nodos hijos. Para los operadores con dos operandos el hijo central (Hc) estará a nulo. Por utilidad también almacenamos el número de nodos del árbol y la profundidad. Para ello utilizamos la siguiente representación:

```
tipo TArbol =  registro{
 Cadena dato;   // operando u operador
 TArbol Hi;     // hijo izquierdo
 TArbol Hc;     // hijo central
 TArbol Hd;     // hijo derecho
 entero num_nodos;     // número de nodos
 entero profundidad; // profundidad del árbol
};
```

2.2 Generación de la población inicial

Se han implementado dos tipos de inicializaciones para los árboles en función de la profundidad máxima especificada como parámetro. En el primer método se generan todas las ramas del árbol hasta la profundidad máxima con nodos que contienen como expresión una función y sólo en el último nivel (hojas) se sitúan los terminales. En la otra alternativa se pueden incluir terminales aunque no se haya alcanzado la profundidad máxima; la probabilidad de elegir un terminal en un nodo concreto crece en función de la profundidad.

En ambos casos los árboles generados de forma aleatoria estarán formados por operadores y operandos. Al crear los árboles se podrían imponer restricciones a los programas para que no fueran demasiado ilógicos o incluso favorecer la creación de programas más lógicos.

Por ejemplo, el programa (*SIC IZQUIERDA DERECHA*) no tiene demasiado sentido y podría ser aconsejable forzar a que el primer operando sea

AVANZA, para que la hormiga alcance la comida que tiene delante. Ciertos programas que a priori no tienen sentido pueden aportar riqueza genética que puede ayudar a la evolución. Las restricciones que se van a aplicar son las siguientes:

- Los árboles generados deben ser correctos: un operador que requiere tres operandos debe tener tres operandos, y uno que requiere dos debe tener dos. Así mismo, los operandos sólo pueden ser hojas del árbol.

- La profundidad de los árboles se limitará mediante un parámetro de entrada para evitar estructuras demasiado grandes que implican un proceso con alto coste a nivel computacional. El operador de cruce tiende a hacer crecer los árboles y puede hacer que el tiempo de evaluación sea excesivo. Por ello, se buscará el valor adecuado de profundidad para que el programa represente una buena estrategia y no afecte demasido al rendimiento del algoritmo.

A continuación se muestra un esquema de función que genera un árbol con una estrategia de rastreo manteniendo las restricciones anteriores y evitando que los árboles tengan una sola hoja:

```
funcion construirArbol(TArbol arbol, entero prof_min, entero
prof_max)  {
   si prof_min > 0  entonces //no puede ser hoja

// generación del subárbol de operador
 operador = operador_aleatorio;

// símbolo de operador aleatorio
 arbol.dato = operador;

 // se generan los hijos
 HI = construir_arbol(arbol.HI, prof_min - 1, prof_max - 1);
 arbol.num_nodos = arbol.num_nodos + arbol.HI.num_nodos;
 si tres_operandos(operador)  entonces
      HC = construir_arbol(arbol.HC,prof_min-1, prof_max-1);
      arbol.num_nodos = arbol.num_nodos + arbol.HC.num_nodos;
 eoc // dos operandos

  HC = NULL;
  HD = construir_arbol(arbol.HD, prof_min - 1, prof_max - 1);
  arbol.num_nodos = arbol.num_nodos + arbol.HD.num_nodos;
 eoc // prof_min = 0
  si prof_max = 0  entonces // sólo puede ser hoja

      // generación del subárbol de operando
      operando = operando_aleatorio;

      // símbolo de operando aleatorio
```

```
        arbol.dato = operando;
        arbol.num_nodos = arbol.num_nodos + 1;
   eoc

   // se decide aleatoriamente operando u operador
   tipo = aleatorio_cero_uno;
   si tipo = 1  entonces // se genera operador
       // generación del subárbol de operador
        { ... }
       eoc // se genera operando
       // generación del subárbol de operando
        { ... }
}
```

Según esta función, cuando se alcanza la profundidad máxima, se genera una hoja correspondiente a un operando. Si no se ha alcanzado la profundidad máxima, se decide aleatoriamente entre un operador y un operando y se genera el árbol que corresponda.

2.3 Función de adaptación

La adaptación o aptitud de un individuo es la cantidad de alimento comido por la hormiga dentro de un espacio de tiempo razonable al ejecutar el programa a evaluar. Se considera que cada operación de movimiento o giro consume una unidad de tiempo. En nuestra versión del problema limitaremos el tiempo a 400 pasos. Lo planteamos como un problema de maximización. Un posible esquema de función de adaptación es el siguiente:

```
funcion adaptacion(TIndividuo individuo, TMapa mapa) {
para cada i desde 0 hasta número filas hacer
  para cada j desde 0 hasta número de columnas hacer
      individuo.matriz[i][j]= mapa[i][j];
      pasos = 0;
      bocados = 0;
      posicionX = 0;
      posicionY = 0;
      direccion = dir.ESTE;
      mientras (pasos < 400 y bocados < 90){
                  ejecutaArbol(individuo.arbol);
      }
adaptacion = bocados;
}
```

La parte principal de la evaluación la realiza la función ejecutaArbol, que analiza el programa codificado en el árbol de un individuo sobre el tablero, que obviamente se va actualizando a medida que va avanzando la hormiga.

```
función ejecutaArbol(TArbol A){
   //mientras no se haya acabado el tiempo ni la comida
     si (pasos < 400 && bocados <90) entonces {

  //si estamos encima de comida comemos
        si (matriz[posicionX][posicionY]== 1) entonces {
           matriz[posicionX][posicionY] = 0;
               bocados++;
           }

  //acciones a realizar en función
  //del nodo en el que estemos
        si A.getValor()== "PROGN3"){
           ejecutaArbol(A.getHi());
           ejecutaArbol(A.getHc());
           ejecutaArbol(A.getHd());
                 }
        eoc si (A.getValor()== "PPROGN2"){
           ejecutaArbol(A.getHi());
           ejecutaArbol(A.getHc());
                 }
        eoc si(A.getValor()== "SIC"){
                 si (hayComida()) ejecutaArbol(A.getHi());
                       eoc ejecutaArbol(A.getHc());
                 }
        eoc si A.getValor()== "AVANZA") Avanza();
         eoc si A.getValor()== "DERECHA") Derecha();
          eoc si(A.getValor()== "IZQUIERDA") Izquierda();
           }
    }
```

En la función se analiza el tipo de nodo en el que estamos y se actúa en consecuencia, ejecutando el subárbol izquierdo, derecho y central en caso de que sea necesario. En caso de que el nodo sea SIC, se llama a la función que comprueba si hay comida en la posición siguiente. Esta función tiene en cuenta que la estructura es toroidal y debe gestionar correctamente los índices de la matriz original para tratarlo como tal:

```
funcion hayComida(): booleano {
  entero x,y = 0;
  si (direccion == dir.ESTE) entonces y = 1;
       eoc si (direccion == dir.OESTE) entonces y = -1;
           eoc si (direccion == dir.NORTE) entonces x = -1;
               eoc x = 1;
       x = x+posicionX;
       si (x = 32) entonces x = 0;
       si (x = -1) entonces x = 31;
       y = y + posicionY;
       si (y== 32) entonces y = 0;
       si (y == -1) entonces y = 31;
       si matriz[x,y]==1 entonces devolver cierto
           eoc devolver falso
```

El resto de funciones se limitan a ejecutar las acciones correspondientes a los operandos o terminales:

```
función avanza(){
 entero x,y = 0;
  si (direccion = dir.ESTE) y = 1;
    eoc si (direccion = dir.OESTE) y = -1;
     eoc si (direccion = dir.NORTE) x = -1;
       eoc x = 1;
  posicionX=posicionX+x;
  si (posicionX = 32) entonces posicionX = 0;
  si (posicionX = -1) entonces posicionX = 31;
  posicionY = posicionY + y;
  si (posicionY = 32) entonces posicionY = 0;
  si (posicionY = -1) entonces posicionY = 31;
  pasos=pasos+1;


función derecha(){
si (direccion = dir.ESTE) entonces direccion = dir.SUR
 eoc
  si (direccion = dir.OESTE) entonces direccion=dir.NORTE
   eoc
    si (direccion=dir.NORTE)  entonces direccion=dir.ESTE
     eoc
      direccion = dir.OESTE;
pasos=pasos+1;
}


función izquierda(){
si (direccion = dir.ESTE) entonces direccion = dir.NORTE
 eoc
  si (direccion = dir.OESTE) entonces direccion = dir.SUR
   eoc
    si (direccion = dir.NORTE)  entonces direccion=dir.OESTE
     eoc
     direccion = dir.ESTE;
 pasos=pasos+1;
}
```

2.4 El operador de cruce

El operador de cruce más utilizado en este tipo de problemas es el cruce por intercambio de subárboles: seleccionamos 2 nodos de manera aleatoria e intercambiamos sus subárboles. Para elegir el nodo de corte podemos utilizar diferentes métodos. Uno de los métodos utiliza diferentes probabilidades de selección según el tipo de nodo.

Por ejemplo, probabilidad de 0.9 si el nodo es un operador y probabilidad 0.1 si el nodo es un operando. Otro método consiste en elegir aleatoriamente un nodo cualquiera *nodoCruce* que exista en ambos padres. Se copian los árboles padre en los hijos y se extrae el subárbol bajo el nodo *nodoCruce*. Después se intercambian los subárboles y se calculan las adaptaciones de los hijos. Un posible esquema de este algoritmo es el siguiente:

```
TArbol subarbol1, subarbol2;
entero num_nodos;


numNnodos=min(numNodos(padre1.arbol),numNodos(padre2.arbol));
nodo_cruce = alea_entero(1,num_nodos);
hijo1.arbol = padre1.arbol;
hijo2.arbol = padre2.arbol;
subarbol1 = hijo1.arbol.BuscarNodo(nodo_cruce);
subarbol2 = hijo2.arbol.BuscarNodo(nodo_cruce);
hijo1.arbol.SustituirSubarbol(nodo_cruce, subarbol2);
hijo2.arbol.SustituirSubarbol(nodo_cruce, subarbol1);
hijo1.adaptacion = adaptacion(hijo1);
hijo2.adaptacion = adaptacion(hijo2);
```

La función `BuscarNodo(n)` devuelve el subárbol que se encuentra por debajo del nodo que ocupa la posición `n` del árbol. La función `SustituirSubarbol(n,a)` sustituye el subárbol que se encuentra bajo el nodo que ocupa la posición `n` por el subárbol `a`. Se puede limitar la profundidad de los árboles generados impidiendo los cruces que dan lugar a árboles demasiado grandes o también se puede penalizar la adaptación de los que sean demasiado grandes.

Si queremos evitar que el operador de cruce genere individuos excesivamente grandes podemos podar el árbol cuando se supere una altura máxima de árbol permitida. Otra opción válida consiste en limitar los árboles a un determinado número de nodos, de manera que la poda se activa cuando el árbol supere ese número.

En realidad lo que estamos limitando es la longitud de los programas. El tamaño de los programas se mide en número de instrucciones, que se puede considerar un valor más importante que la profundidad máxima del árbol. Si tenemos en cuenta el objetivo de que la hormiga encuentre la comida, las restricción en altura impide concatenar más de un número determinado de funciones y el espacio de soluciones se restringe considerablemente.

Por ejemplo, si fijamos el número máximo de instrucciones de un programa a 40, dejamos a los programas que adquieran la longitud de rama que necesiten pero como máximo a 40.

2.5 Operador de mutación

Un posible operador de mutación es el método que muta un nodo del árbol elegido al azar: si el nodo es un terminal cambiamos su valor por otro terminal, y si es una función cambiamos su valor por el de otra función. Si el nodo corresponde a una función, debemos sustituir de manera aleatoria la función existente en ese nodo por otra, teniendo en cuenta el número de operandos de la función original. Por ejemplo, si la función era SIC se cambia por PROGN2.

```
booleano mutado;
entero i,nodo_mut;
real prob;
TArbol subarbol;
para cada i  desde 0  hasta tam_pob  hacer{
    mutado =  falso;
    // se genera un número aleatorio en [0 1)
    prob = alea();
    si (prob < prob_mut) entonces  {
      mutado =  cierto;
      // se selecciona un nodo
      nodo_mut = alea_entero(1,num_nodos(pob[i].arbol));
      subarbol = pob[i].arbol.BuscarNodo(nodo_mut);
      si no EsHoja(subarbol)  entonces
                CambiarOperador(subarbol)
      eoc
          CambiarOperando(subarbol)
  }
si (mutado)
    pob[i].adaptacion = adaptacion(pob[i], lcrom);
    . . .
```

Esta función elige, en caso de mutar según la probabilidad de mutación, uno de los nodos del árbol aleatoriamente y lo cambia según sea operador u operando. La función *CambiarOperando* cambia el operando por otro distinto elegido aleatoriamente. Así mismo, la función *CambiarOperador* cambia el operador por otro distinto elegido aleatoriamente, pero en este caso puede ser necesario modificar el árbol para respetar el número de argumentos requerido por cada operador:

- Si el nuevo operador tiene tres argumentos y el antiguo tenía dos, se genera de forma aleatoria un nuevo árbol como hijo central (HC).

- Si el nuevo operador tiene dos argumentos y el antiguo tenía tres, se elimina el hijo central (HC).

- En cualquiera de los dos casos anteriores es necesario volver a calcular el número de nodos del árbol.

Un método de mutación que genera buenos resultados es el de mutación de árbol. Con este método seleccionamos al azar un nodo y generamos de forma aleatoria un nuevo subárbol por debajo de ese nodo. Bastaría con cambiar parte del código anterior:

```
. . .
subarbol = pob[i].arbol.BuscarNodo(nodo_mut);
si no EsHoja(subarbol) entonces
     construirArbol(subarbol,nuevaprofundidad)
eoc
. . . .
```

También se han implementado otros métodos de mutación independientes según el tipo de nodo:

- Mutación de Terminales: seleccionamos al azar un nodo terminal (una hoja) del árbol y sustituimos de manera aleatoria el terminal existe en ese nodo por otro. Por ejemplo, si el terminal era AVANZA se cambia por DERECHA o IZQUIERDA.

- Mutación Funcional: seleccionamos al azar un nodo que no sea una hoja del árbol y sustituimos de manera aleatoria la función existente en ese nodo por otra con la misma aridad. Por ejemplo, si la función era SIC se cambia por PROGN2.

3. UN EJEMPLO DE INTERFAZ GRÁFICA

Para finalizar el análisis del proyecto mostramos un ejemplo de una aplicación que incluye una interfaz gráfica que permite seleccionar diferentes parámetros para el algoritmo, así como visualizar el programa obtenido y la ruta seguida por la hormiga utilizando la estrategia codificada en dicho programa.

En la figura 4 se muestra la interfaz gráfica de la aplicación, que se explica más adelante.

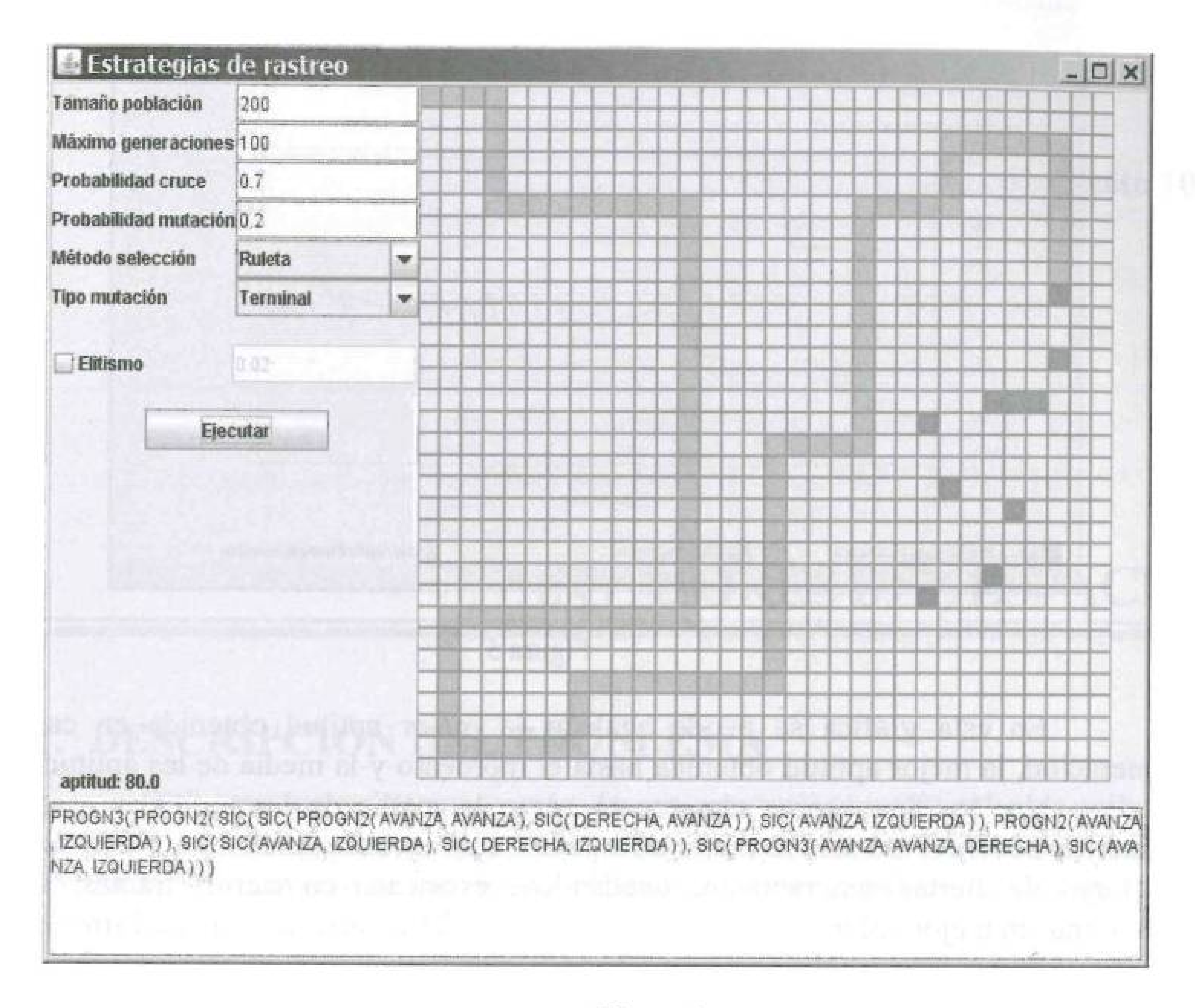


Figura 4

Como se puede observar en la figura anterior, la aplicación permite seleccionar los parámetros básicos del algoritmo: tamaño de la población, número de generaciones, método de selección (Ruleta, Torneo o Ranking), probabilidad de cruce y mutación y tipo de mutación (de árbol, de terminal, de función o de ambas). También permite ejecutar el algoritmo en su versión con elitismo.

Al finalizar la ejecución del algoritmo genético podemos observar el programa obtenido y la ruta seguida por la hormiga. La aplicación también permite analizar la evolución mediante gráficas como la siguiente:

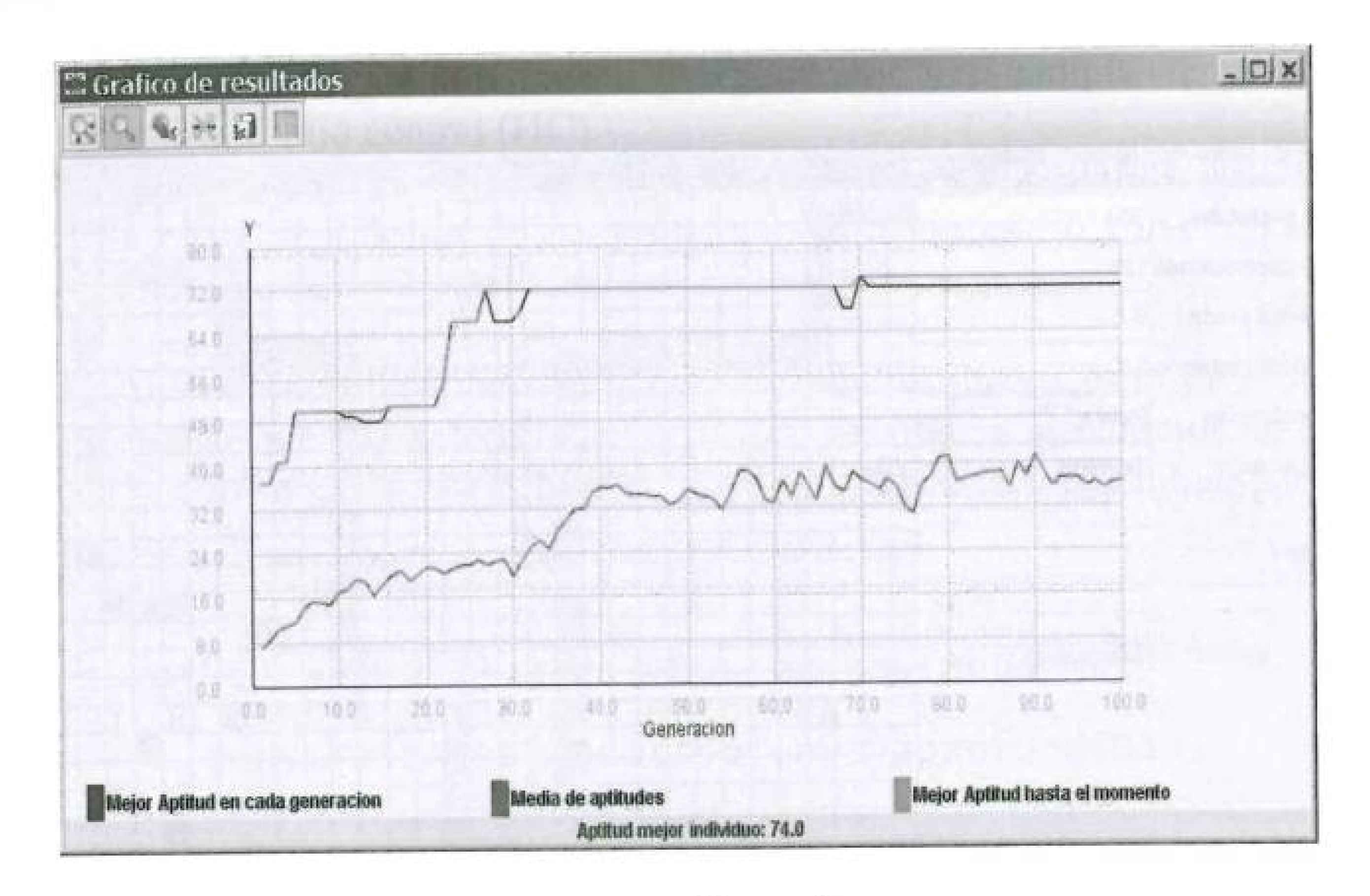


Figura 5

En esta gráfica se puede analizar la mejor aptitud obtenida en cada generación, la mejor aptitud obtenida hasta el momento y la media de las aptitudes de la población. Se observa claramente cómo la media de las aptitudes crece. También podemos ver que la mejor aptitud que hay en cada generación se repite a lo largo de ciertas generaciones, quedándose estancada en ciertos tramos, sin encontrar un mejor valor.

Proyecto 10

INVASORES DEL ESPACIO

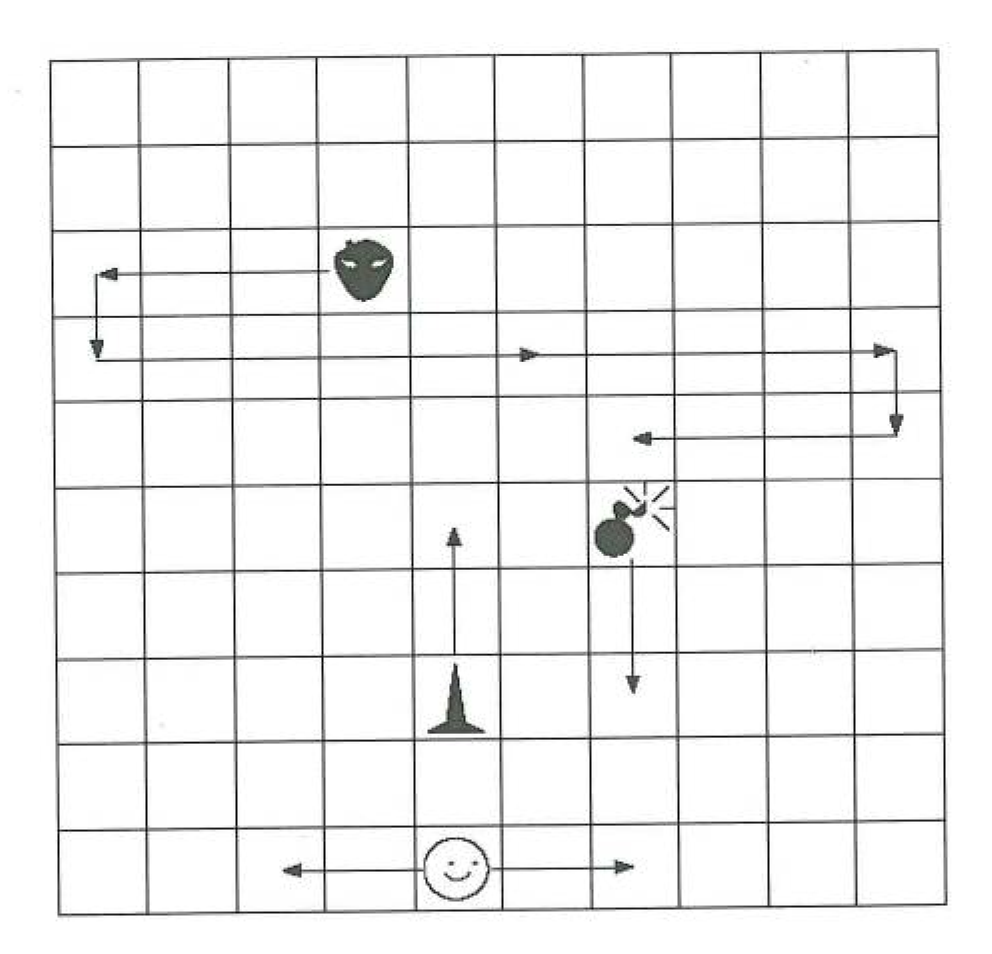
1. DESCRIPCIÓN DEL PROBLEMA

En este proyecto usaremos la programación genética para obtener mediante evolución estrategias de defensa para el juego de los INVASORES DEL ESPACIO. El proyecto se basa en los experimentos de un trabajo de investigación desarrollado por Jackson [JAC05].

En este juego hay un único defensor en la parte más baja de la pantalla, y una serie de invasores del espacio que descienden del cielo, dejando caer bombas a medida que se mueven, como muestra la figura, aunque, el tamaño de la pantalla de juego será de 20x20, en lugar de 10x10.

El objetivo del defensor es disparar a los alienígenas antes de que aterricen, a la vez que evita que le destruyan las bombas. Para conseguir esto, el defensor sólo tiene tres acciones disponibles: moverse a la izquierda, moverse a la derecha y disparar un misil.

Nosotros consideraremos inicialmente una versión simplificada con un único alienígena, que no lanza bombas, y un único defensor.



En Lisp los programas, a parte de la asignación de valores a variables y la salida de resultados, son una función de un conjunto de operandos y operadores. El conjunto de operandos (terminales) para este problema es:

$$T = \{IZQUIERDA, DERECHA, FUEGO, DIST_Y, DIST_X\}$$

IZQUIERDA y DERECHA mueven al defensor una celda en la dirección correspondiente, siempre que no se encuentre en el límite del tablero que haga imposible el movimiento. FUEGO lanza un misil desde las coordenadas actuales del defensor, a menos que ya haya un misil en el aire. La ejecución de nodos IZQUIERDA, DERECHA y FUEGO durante la evaluación de la adaptación hace que el juego avance un paso.

Estos tres nodos devuelven cero como resultado de la función de evaluación del árbol. DIST_Y devuelve la distancia vertical actual del alienígena a la base del tablero y DIST_X, la distancia horizontal al defensor. El valor de DIST_X es positivo si el alienígena se está acercando al defensor, y negativo si se está alejando.

Un posible conjunto de funciones para este problema es el siguiente:

$$O = \{IF, EQ, PROGN2, PROGN3\}$$

La función IF toma tres argumentos y se define como

```
IF <arg1> then
            <arg2>
          else
            <arg3>
```

La función EQ evalúa sus dos argumentos y devuelve 1 si son iguales y 0 en otro caso. En el conjunto de operadores incluiremos también conectivas que fuerzan la ejecución de sus argumentos en un determinado orden. Tomando el nombre de una conectiva de Lisp, PROGN, que causa este efecto, incluimos en el conjunto de operadores las conectivas PROGN2 y PROGN3, que toman dos y tres argumentos respectivamente. Con estos conjuntos de generan individuos como el siguiente:

$$
\begin{aligned}
(IF \quad & (EQ \qquad (DIST_Y \; DIST_X)) \\
& (PROGN3 \quad (FUEGO \; IZQUIERDA \; DIST_X)) \\
& (PROGN2 \quad (IZQUIERDA \; IZQUIERDA)))
\end{aligned}
$$

Las expresiones de este tipo pueden representarse como árboles de análisis, compuestos de funciones y terminales.

Un posible algoritmo de programación genética para este problema tiene las siguientes características:

- Se genera un población inicial de programas aleatorios (árboles) usando el conjunto de funciones y terminales posibles. Estos árboles, que deben ser sintácticamente correctos, tendrán distintos tamaños, aunque conviene establecer un límite a su profundidad.

- Para evaluar la adaptación de un individuo, el código del programa se ejecuta sobre un conjunto de juegos aleatorios (por ejemplo, 50 juegos). Para cada prueba del programa candidato, el defensor se coloca aleatoriamente en algún punto de la fila inferior del tablero. El alienígena se materializa aleatoriamente en cualquiera de las 6 filas superiores, y viaja en una dirección elegida aleatoriamente. El alienígena desciende en zig-zag, como muestra la figura, cruzando el tablero hasta llegar al extremo y bajando a la siguiente fila, que se recorre en la dirección contraria a la anterior. Sólo puede haber un misil simultáneamente en el aire. El defensor debe esperar hasta que el misil

en curso da sobre el alienígena o se sale del tablero, antes de lanzar el siguiente.

- El juego avanza en pasos. La ejecución de las primitivas IZQUIERDA, DERECHA y FUEGO hacen que el tiempo avance una unidad. Los otros nodos del árbol de un programa pueden verse como un tiempo de "toma de decisiones" que se considera despreciable, y no hacen que avance el tiempo.

- En cada paso de tiempo, el alienígena y el misil avanzan una celda en sus direcciones respectivas. Al no haber instrucciones iterativas entre los operadores, cada programa se evalúa repetidamente hasta que el juego termina.

- Para permitir el avance de programas que "sólo piensan", es decir, que no tienen primitivas IZQUIERDA, DERECHA o FUEGO, el tiempo también avanza una unidad al final de cada ejecución de estos programas.

- Un juego termina cuando el alienígena es destruido o cuando consigue aterrizar.

- Podemos definir la función de adaptación como una medida de lo cerca que llegan los misiles del defensor de dar al alienígena. Cuando un misil alcanza el mismo valor del eje-Y que el alienígena, la distancia considerada es la diferencia absoluta de las coordenadas X. La suma de estas distancias se registra como el resultado del juego correspondiente. Los valores de las distancias se suman para 50 juegos, para dar un valor final de adaptación. Se introduce también una penalización adicional (200 puntos) que se añade a la adaptación cada vez que el alienígena consigue aterrizar.

- Una ejecución se considera un éxito si produce un programa que gane los 50 juegos. La adaptación de este programa puede ser o no ser cero.

- Se aplican los operadores de selección, cruce y mutación para producir un cierto número de generaciones con una elección adecuada de parámetros para el algoritmo.

2. DISEÑO DEL ALGORITMO

Antes de considerar el diseño de los distintos componentes del algoritmo vamos a ocuparnos de la representación del espacio en el que se van a mover el defensor, el alienígena, los misiles y las bombas. Es decir, necesitamos representar la cuadrícula o tablero en el que tienen lugar los movimientos, y el estado y posición de los ocupantes.

El tablero será simplemente una matriz cuyas casillas pueden estar libres o contener uno de los posibles ocupantes:

```
tipo TTablero: matriz[max_x,max_y] de TEstado;

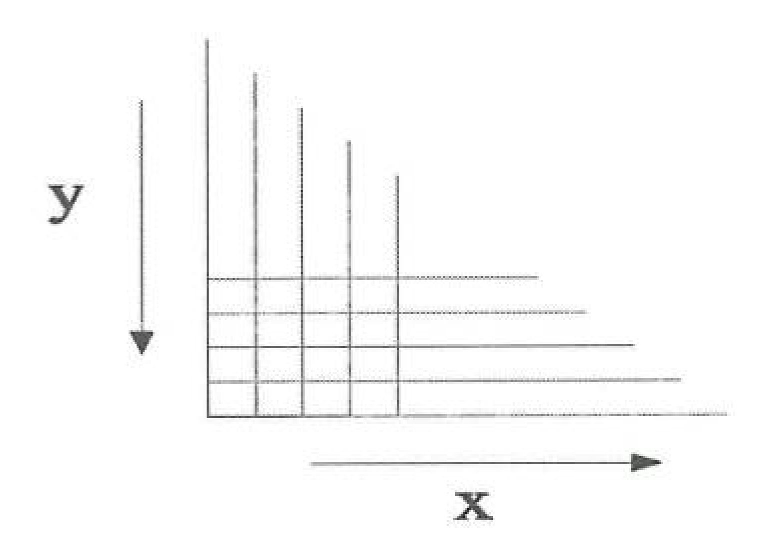
tipo TEstado: {libre, misil, alien, defensor};
```

La situación de cada ocupante, que vamos a representar con el tipo *TOcupa*, está formada por las coordenadas que ocupa en el tablero, y por su dirección de movimiento, Este u Oeste. Este información sólo tiene sentido para el defensor y el alien, ya que los misiles y bombas sólo se mueven verticalmente.

```
tipo TOcupa = registro{
   entero x; // coordenada x
   entero y; // coordenada y
   TDir dir; // función de evaluación
};

tipo TDir: {este, oeste};
```

Adoptamos el convenio que muestra la siguiente figura para el crecimiento de la x y la y que representan una coordenada del tablero:



Es decir, el alien comienza en 1 y cuando baja la *y* va aumentando. Al misil le ocurre lo contrario.

Finalmente, podemos agrupar toda esta información en un tipo *TEspacio* que incluye los datos del tablero y sus ocupantes. Parte de esta información es redundante. Por ejemplo, recorriendo todas las casillas del tablero podemos comprobar si hay algún misil y dónde se encuentra. Pero al representarla explícitamente, hacemos más eficiente la ejecución, lo que es muy importante en un algoritmo de este tipo.

```
tipo TEspacio = registro{
   TTablero tab; // cuadrícula de movimientos
   booleano hay_misil; // Indicativo de si hay un
                       // misil en el tablero
   TOcupa defensor; // situación del defensor
   TOcupa misil; // situación del misil
   TOcupa alien; // situación del alienígena
};
```

El programa principal del algoritmo sigue el esquema general de una minimización, como el descrito para el proyecto 1. Pero en lugar de trabajar con cadenas binarias trabaja con árboles, y por tanto con diferentes operadores.

Vamos a considerar ahora cada uno de los aspectos de este algoritmo de programación genética.

2.1 Representación de los individuos y la población

La población continúa siendo una colección de individuos, como en algoritmos anteriores:

```
tipo TPoblacion: vector de TIndividuo;
```

Los individuos están formados, además de por los datos que se requieran para el algoritmo evolutivo, por un árbol que es la representación sintáctica de una función.

```
tipo TIndividuo = registro{
   TArbol arbol; //estrategia de juego
   real adaptación;//función de evaluación
   real puntuacion;//puntuación relat:adaptación/sumadapt.
   real punt_acu; //puntuación acumulada para sorteos
   booleano elite;//elitismo
};
```

Los árboles están formados por una combinación de nodos internos correspondientes a los operadores (IF, EQ, PROGN2, PROGN3) y de nodos terminales u hojas correspondientes a los operandos (IZQUIERDA, DERECHA, FUEGO, DIST_Y, DIST_X).

Estos árboles representa la estrategia de juego del defensor, que debe moverse de forma que sus misiles maten al alienígena. Esta estrategia puede aplicarse a las sucesivas configuraciones del tablero que representa el espacio de la invasión.

La representación del árbol de estrategia de juego debe tener en cuenta que hay operadores de dos y de tres operandos. Una posible representación es la siguiente:

```
tipo TArbol = registro{
   cadena_caracteres dato; // operando u operador
   TArbol HI; // hijo izquierdo
   TArbol HC; // hijo central
   TArbol HD; // hijo derecho
   entero num_nodos; // número de nodos
};
```

Para los operadores que sólo tienen dos operandos, el hijo central HC del árbol toma el valor nulo (NULL). También es útil para la eficiencia del algoritmo almacenar el número total de nodos del árbol, lo que evita tener que calcularlo cuando se necesita.

2.2 Generación de la población inicial

La población inicial está compuesta de árboles generados aleatoriamente, es decir, con combinaciones aleatorias de los operadores y operandos. Algunas combinaciones no tendrán sentido, pero no importa, ya que introducen riqueza genética que puede ser útil en la evolución. En este proyecto, si se fuerza demasiado la forma de los posibles árboles, el resultado empeora. Las restricciones que aconsejamos introducir son las siguientes:

- Los árboles generados deben ser sintácticamente correctos: un operador que requiere tres operandos debe tener tres operandos, y uno que requiere dos debe tener dos. Así mismo, los operandos sólo pueden ser hojas del árbol.

- Lo que sí necesitamos limitar para que el algoritmo sea computacionalmente viable es la profundidad del árbol. Si permitimos

que los árboles tengan cualquier tamaño, normalmente crecerán hasta hacer que el tiempo necesario para su evaluación sea excesivo. El límite a la profundidad del árbol será un parámetro de entrada seleccionado por el usuario. A mayor profundidad permitida en los árboles, más lento será el algoritmo, pero se tendrá mayor riqueza genética, que puede permitir encontrar estrategias más sofisticadas. Hay que buscar el equilibrio adecuado.

También es útil imponer una profundidad mínima que nos sirva, por ejemplo, para especificar que los árboles no puedan ser sólo una hoja.

Veamos un posible esquema de la función que construye un árbol de estrategia teniendo en cuenta las restricciones que hemos mencionado.

```
funcion construir_arbol(TArbol arbol, entero prof_min,
                        entero prof_max)
{
   si prof_min > 0 entonces{ //no puede ser hoja
      // generación del subárbol de operador
      operador = operador_aleatorio;

      // símbolo de operador aleatorio
      arbol.dato = operador;

      // se generan los hijos
      HI=construir_arbol(arbol.HI,prof_min-1, prof_max-1);
      arbol.num_nodos=arbol.num_nodos + arbol.HI.num_nodos;
      si tres_operandos(operador)  entonces{
       HC=construir_arbol(arbol.HC,prof_min-1, prof_max-1);
       arbol.num_nodos=arbol.num_nodos +arbol.HC.num_nodos;
      }
      eoc // dos operandos
      HC = NULL;
      HD=construir_arbol(arbol.HD, prof_min-1,prof_max-1);
      arbol.num_nodos=arbol.num_nodos + arbol.HD.num_nodos;
   }
   eoc{ // prof_min = 0
      si prof_max = 0 entonces{
         // sólo puede ser hoja
         // generación del subárbol de operando
         operando = operando_aleatorio;
         arbol.dato = operando;
         arbol.num_nodos = arbol.num_nodos + 1;
      }
      eoc{
         // se decide aleatoriamente operando u operador
         tipo = aleatorio_cero_uno;
         si tipo = 1 entonces // se genera operador
            // generación del subárbol de operador
                     { ··· }
```

```
          eoc // se genera operando
              // generación del subárbol de operando
                      { ... }
        }
    }
}
```

Esta función comprueba si en la construcción del árbol ya se ha alcanzado la profundidad mínima fijada. Si no se ha alcanzado esta profundidad mínima, el subárbol a construir debe tener como nodo raíz un operador. Si se ha alcanzado la profundidad mínima, entonces comprobamos la profundidad máxima. Si se ha alcanzado la profundidad máxima, sólo podemos generar una hoja, es decir, el subárbol correspondiente a un operando. Si no se ha alcanzado la profundidad máxima, se decide aleatoriamente si el subárbol corresponde a un operando o a un operador, y se genera lo que corresponda.

2.3 Función de adaptación

La adaptación de un individuo es la suma del resultado de aplicar la estrategia de juego representada por su árbol a un cierto número de partidas NUM_PARTIDAS (por ejemplo, 50). El resultado de cada partida es la suma de la distancia del misil al alien cada vez que se encuentran en la misma fila. Se trata por tanto de una **minimización**.

Un posible esquema de la función de adaptación es el siguiente:

```
funcion adaptacion(TIndividuo individuo)
{
   TEspacio espacio; // espacio de juego
   booleano gana_usuario;
   // indicativo del resultado de la partida

   entero np; // contador de partidas
   entero tiempo, tiempo_anterior; // contadores de tiempo
   entero distancia, nueva_distancia;
   // contador de distancias
   para np = 0 hasta NUM_PARTIDAS hacer{
     generar_partida(espacio);
     tiempo = 0;
     distancia = 0;
     gana_usuario = falso;
     mientras no Terminacion(espacio, gana_usuario)  hacer{
         tiempo_anterior = tiempo;
         evaluar(individuo.arbol, espacio, tiempo);
         si tiempo_anterior = tiempo entonces{
          tiempo = tiempo + 1;
          avanzar_tiempo(espacio,termina,nueva_distancia);
```

```
            distancia = distancia + nueva_distancia;
          }
        }
        adaptacion = adaptacion + distancia;
    }
    si no gana_usuario entonces
        adaptacion = adaptacion + PENALIZA_ALIEN;
   devolver adaptacion;
}
```

En cada partida de las jugadas se comprueba si ha habido avance del tiempo, y si no ha sido así se le hace avanzar una unidad. Se generan de forma aleatoria NUM_PARTIDAS mediante la función *generar_partida*:

```
funcion generar_partida(var TEspacio espacio)
{
     // se inicializa el tablero a libre
                    ...
     espacio.hay_misil = falso;
     // se genera aleatoriamente la posición del defensor
     espacio.defensor.x = alea_entero(1,max_x);
     espacio.defensor.y = max_y;
     // posición del alien aleatoria,
     // entre las 6 primeras filas
     espacio.alien.x = alea_entero(1,max_x);
     espacio.alien.y = alea_entero(1,6);
     // dirección de movimiento aleatoria
     espacio.alien.dir = alea_valor(ESTE,OESTE);
}
```

Con el transcurso del tiempo, el alien avanza en la dirección de su movimiento, bajando una casilla cuando llega al extremo del tablero. También el misil se mueve, pero en su caso subiendo una casilla. Representamos este avance por la función *avanzar_tiempo*. Si el alien y el misil se cruzan, la función calcula la distancia entre ellos en ese momento.

```
funcion avanzar_tiempo(var TEspacio espacio, entero
     distancia)
{
    // avance del alien
  si espacio.alien.dir = ESTE entonces{

     espacio.tablero[espacio.alien.x,espacio.alien.y]=libre;
     // se comprueba si está en el extremo del tablero
     si espacio.alien.x < max_x entonces
          espacio.alien.x = espacio.alien.x + 1;
     eoc{ //extremo del tablero
          // baja y cambia la dirección a OESTE
```

```
          espacio.alien.dir = OESTE;
          espacio.alien.y = espacio.alien.y + 1;
       }

     espacio.tablero[espacio.alien.x,espacio.alien.y]=alien;
    }
  eoc //espacio.alien.dir = OESTE
            { análogo }

    //avance del misil
  si espacio.hay_misil = cierto entonces{
   si espacio.misil.y > 0 entonces
  espacio.tablero[espacio.misil.x,espacio.misil.y]=libre;
  espacio.misil.y = espacio.misil.y - 1;
  espacio.tablero[espacio.misil.x,espacio.misil.y]=misil;
  si espacio.misil.y = espacio.alien.y y
              espacio.misil.x ≠ espacio.alien.x entonces
    distancia = abs(espacio.misil.x - espacio.alien.x);
    }
   eoc{
   espacio.tablero[espacio.misil.x, espacio.misil.y]=libre;
   espacio.hay_misil = falso;
   }
  }
}
```

La función *Terminación* comprueba si se ha llegado al final del juego, bien porque el alien haya aterrizado, o bien por que haya sido alcanzado por el misil.

```
booleano Terminacion(TEspacio espacio, booleano
    gana_usuario)
{
   booleano termina;
   termina = falso;

   si (espacio.alien.y = max_y) entonces // el alien ha
    aterrizado
      gana_usuario = falso;
      termina = cierto;
   eoc si (espacio.hay_misil y
          (espacio.misil.x = espacio.alien.x)
           y (espacio.misil.y = espacio.alien.y))
      gana_usuario = cierto;
      termina = cierto;
   devuelve termina;
}
```

La evaluación del árbol consiste en aplicar la estrategia que recoge dicho árbol a la partida que corresponda. La función de evaluación del árbol es una función recursiva que aplica el operando u operador que corresponda. Un esquema simplificado de dicha función es el siguiente:

```
funcion evaluar(TArbol arbol, entero tiempo,
                 entero distancia,booleano termina)
   entero valor1, valor2, valor3; // auxiliares
   entero nueva_distancia; // auxiliar
   valor1=0; valor2=0; valor2=0; //inicializaciones

   si EsHoja(arbol) entonces{
     si dato = FUEGO entonces{
       si espacio.hay_misil = falso entonces{
        espacio.hay_misil = cierto;
      // misil en la posición superior al defensor
      espacio.misil.x = espacio.defensor.x;
      espacio.misil.y = espacio.defensor.y + 1;
      espacio.tablero[espacio.misil.x,espacio.misil.y]=misil;
      }
      tiempo = tiempo + 1;
      avanzar_tiempo(espacio, termina, nueva_distancia);
      distancia = distancia + nueva_distancia;
      }
      eoc si dato = IZQUIERDA entonces{
         si espacio.defensor.x > 0 entonces
            espacio.defensor.x = espacio.defensor.x - 1;
         tiempo = tiempo + 1;
         avanzar_tiempo(espacio, termina, nueva_distancia);
         distancia = distancia + nueva_distancia;
      }
      eoc si dato = DERECHA entonces{
               { ··· } // simétrico
      eoc si dato = DIST_Y entonces
         valor1 = espacio.alien.y;
      eoc si dato = DIST_X entonces{
         valor1 = abs(espacio.alien.x - espacio.defensor.x);
         si se alejan entonces
       //comprobar en función de posiciones y
      // dirección del movimiento

         valor1 = - valor1;
      }
      devolver valor1;
   }
   eoc // es operador
           ...
}
```

La función comprueba si la raíz del árbol que debe evaluar es una hoja o no, es decir, si corresponde a un operando o a un operador. Si es un operando, aplica al espacio, es decir, al tablero y sus ocupantes, las modificaciones que correspondan en función del operando de que se trate, y termina. Si es un operador, hace la evaluación de los subárboles que corresponda. Podemos observar que los operando DIST_X y DIST_Y no suponen avance de tiempo.

Un esquema del resto del código para el caso de los operadores es el siguiente:

```
eoc{
 // es operador
 si dato = IF entonces{
   valor1 = evaluar(arbol.HI, espacio, tiempo, termina);
     si no termina entonces
       si valor1 <> 0 entonces
        devolver evaluar(arbol.HC,espacio,tiempo,termina);
       eoc
        devolver evaluar(arbol.HD, espacio,tiempo,termina);
       eoc
          devolver valor1;
   }
   eoc si dato = PROGN2 entonces{
     valor1 = evaluar(arbol.HI, espacio, tiempo, termina);
     si no termina entonces{
      valor2 = evaluar(arbol.HD, espacio, tiempo, termina);
      devolver valor1 + valor2;
       }
    }
    eoc si dato = PROGN3 entonces{
     valor1 = evaluar(arbol.HI, espacio, tiempo, termina);
      si no termina entonces
       valor2 = evaluar(arbol.HC,espacio,tiempo,termina);
        si no termina entonces
         valor3=evaluar(arbol.HD, espacio,tiempo, termina);
         devolver valor1 + valor2 + valor3;
    eoc si dato = EQ entonces{
       valor1=evaluar(arbol.HI, espacio, tiempo, termina);
       si no termina entonces{
          valor2=evaluar(arbol.HD,espacio,tiempo,termina);
          si valor1 = valor2 entonces
             devolver 1;
          eoc
             devolver 0;
      }
   }
   devolver 0;
}
```

En función del operador de que se trate se evalúan los árboles hijos que sean necesarios. Excepto para la evaluación del primero de los hijos, siempre se comprueba que el juego en curso no haya finalizado en el paso anterior.

2.4 El operador de cruce

Un posible operador de cruce consiste en intercambiar dos subárboles (elegidos aleatoriamente) entre los dos árboles padres.

Para hacer esto se elige aleatoriamente la posición de un nodo *nodo_cruce* que exista en ambos padres. Se copian los árboles padre en los hijos y se extrae el subárbol bajo el nodo *nodo_cruce*. Después se intercambian los subárboles y se calculan las adaptaciones de los hijos. Un posible esquema de este algoritmo es el siguiente:

```
TArbol subarbol1, subarbol2;
entero num_nodos;

num_nodos=minimo(num_nodos(padre1.arbol),
                 num_nodos(padre2.arbol));
nodo_cruce = alea_entero(1,num_nodos);
hijo1.arbol = padre1.arbol;
hijo2.arbol = padre2.arbol;
subarbol1 = hijo1.arbol.BuscarNodo(nodo_cruce);
subarbol2 = hijo2.arbol.BuscarNodo(nodo_cruce);
hijo1.arbol.SustituirSubarbol(nodo_cruce, subarbol2);
hijo2.arbol.SustituirSubarbol(nodo_cruce, subarbol1);
hijo1.adaptacion = adaptacion(hijo1);
hijo2.adaptacion = adaptacion(hijo2);
```

La función *BuscarNodo(n)* devuelve el subárbol bajo el nodo que ocupa la posición *n* del árbol. La función *SustituirSubarbol(n, a)* sustituye el subárbol que se encuentra bajo el nodo que ocupa la posición *n* por el subárbol *a*. La implementación de estas funciones es muy dependiente del lenguaje de programación y de la forma de programación (programación orientada a objetos, memoria dinámica, etc.).

Se puede limitar la profundidad de los árboles generados impidiendo los cruces que dan lugar a árboles demasiado grandes. Una opción más sencilla es penalizar la adaptación de los que sean demasiado grandes. En cualquier caso, si la profundidad de los árboles de la población inicial es reducida, se pueden obtener buenos resultados aun sin limitar la profundidad de los árboles resultantes del cruce.

2.5 El operador de mutación

Un posible operador de mutación consiste en cambiar un operador por otro, teniendo en cuenta que no todos los operadores tienen el mismo número de operandos.

```
booleano mutado;
entero i,nodo_mut;
real prob;
TArbol subarbol;

para cada i desde 0 hasta tam_pob hacer{
   mutado = falso;
   // se genera un número aleatorio en [0 1]
   prob = alea();
   si (prob < prob_mut){
      mutado = cierto;
      // se selecciona un nodo
      nodo_mut = alea_entero(1,num_nodos(pob[i].arbol));
      subarbol = pob[i].arbol.BuscarNodo(nodo_mut);
      si no EsHoja(subarbol) entonces
         CambiarOperador(subarbol)
      eoc
         CambiarOperando(subarbol)
   }
   si (mutado)
      pob[i].adaptacion = adaptacion(pob[i], lcrom);
}
```

Esta función comprueba la tasa de mutación y si se cumple elige uno de los nodos del árbol aleatoriamente, para modificar el árbol que se encuentra bajo dicho nodo. La función *CambiarOperando* cambia el operando por otro distinto elegido aleatoriamente. Así mismo, la función *CambiarOperador* cambia el operador por otro distinto elegido aleatoriamente, pero en este caso puede ser necesario modificar el árbol para respetar el número de argumentos requerido por cada operador:

- Si el nuevo operador tiene tres argumentos y el antiguo tenía dos, se genera de forma aleatoria un nuevo árbol como hijo central (HC).
- Si el nuevo operador tiene dos argumentos y el antiguo tenía tres, se elimina el hijo central (HC).
- En cualquiera de los dos casos anteriores hay que recalcular el número de nodos del árbol.

2.6 Consideraciones adicionales

Es un algoritmo costoso computacionalmente que requerirá bastante tiempo de ejecución. Por ello conviene limitar la profundidad de los árboles de la población inicial, por ejemplo a 4 o a un valor cercano a este. Valores mayores darán mejores resultados pero requerirán más tiempo de ejecución. Valores adecuados para el número máximo de generaciones están entre 50 y 200.

Una buena ayuda para la depuración y evaluación del resultado obtenido es representar gráficamente la evolución de un juego en el tablero al aplicar uno de los programas de la población.

3. UNA VERSIÓN MÁS COMPLEJA: CON BOMBAS

El proyecto puede hacerse más interesante, lo que también incrementa la complejidad de su desarrollo, si el alienígena tiene capacidad de arrojar bombas aleatoriamente según cruza el tablero. Sólo se permite un número máximo de bombas simultáneamente en el aire, que se controla por el parámetro MAX_BOMB (entre 3 y 10). Otra restricción es que las bombas no pueden compartir la misma coordenada X, es decir, sólo puede haber una bomba en cada columna del tablero. Una bomba se mueve hacia abajo una casilla en cada instante de tiempo que avanza el juego. Ahora el juego también puede terminar si cae una bomba sobre el defensor. El defensor no puede disparar a la bomba, sólo puede apartarse de su camino. Para poder saber cuándo necesita hacer esto, el conjunto de terminales se expande de la siguiente forma:

$$T = \{IZQUIERDA, DERECHA, FUEGO, \\ DIST_Y, DIST_X, ATACADO\}$$

El terminal ATACADO devuelve 1 si el defensor está directamente debajo de una bomba y cero en otro caso. Añadimos una penalización de 200 puntos si el defensor es destruido por una bomba. Para implementar esta alternativa debemos incluir en la representación del espacio las bombas:

```
tipo TEstado: { libre, misil, alien, defensor, bomba };
constante entero MAX_BOMBAS = ...
tipo TBombas: vector de TOcupa;
tipo TEspacio = registro{
   TTablero tab; // cuadrícula de movimientos
       ...
   TBombas bombas; // situación de las bombas
};
```

La simulación del avance de tiempo con *AvanzarTiempo* también debe tener en cuenta a los nuevos elementos. En cada instante de tiempo las bombas se mueven una posición hacia abajo, hasta que se salen del tablero, es decir, para cada bomba del vector *bombas* se actualiza su posición. En la función *AvanzarTiempo* también se decide aleatoriamente si el alienígena lanza nuevas bombas. Para ello se comprueba que no se haya alcanzado ya el número máximo de bombas *MAX_BOMBAS* permitidas en el tablero y que no haya ya otra bomba en la casilla del alien. Si se cumplen estas condiciones se decide aleatoriamente si se lanza o no una nueva bomba, y en caso de que se lance se actualiza el espacio con los nuevos datos.

Para implementar esta versión hay que incluir el nuevo operando *ATACADO*. Para ello se incluye en la lista de operandos, desde la que se puede elegir cuando se están creando los árboles. En la función *evaluar* que aplica los operadores y operandos de un árbol al espacio de juego, se añade este operando *ATACADO* que devuelve 1 si alguna de las bombas está en la misma columna que el defensor.

La función *Terminación*, que comprueba si se ha llegado al final del juego, ahora también verifica si alguna de las bombas ha caído sobre el defensor. En dicho caso se llega al final de la partida perdiendo el usuario y se introduce una penalización a la adaptación, por ejemplo de 200 puntos.

4. MÁS DIFÍCIL TODAVÍA: MÚLTIPLES ALIENÍGENAS

En una versión aún más compleja podemos introducir múltiples alienígenas. Esto lleva a modificar el conjunto de terminales y funciones de la siguiente forma:

$$\begin{aligned} T = & \ \{IZQUIERDA, DERECHA, FUEGO, ATACADO, \\ & \ BLANCO_IZQ, BLANCO_DER\} \\ O = & \ \{IF, PROGN2, PROGN3\} \end{aligned}$$

Los terminales BLANCO_IZQ y BLANCO_DER devuelven 1 si un misil lanzado a la izquierda/derecha de la posición actual del defensor consiguiera un disparo mejor que uno lanzado desde la posición actual. En otro caso devuelven 0. La dificultad para obtener soluciones a este problema depende de los valores de MAX_BOMB y de un nuevo parámetro MAX_ALIEN. Consideraremos valores entre 1 y 3 para ambos parámetros.

En esta versión hay que volver a modificar la representación del espacio para incluir más alienígenas.

```
constante entero MAX_ALIEN = 3
tipo TAlien: vector de TOcupa;
tipo TEspacio = registro{
   TTablero tab; // cuadrícula de movimientos
        ...
   TAlien alienigenas; // situación de los alien
};
```

En la función *AvanzarTiempo* avanzan todos los alienígenas del vector correspondiente, siguiendo el movimiento en zig-zag de estas criaturas. Si no se ha llegado al número máximo de alienígenas permitidos en el tablero se decide aleatoriamente si se crea uno nuevo en alguna de las seis primeras filas.

También hay que modificar los conjuntos de operandos y operadores. Se elimina *EQ* de los operadores. De los operandos se eliminan *DIST_X* y *DIST_Y*, y en su lugar se añaden *BLANCO_IZQ* y *BLANCO_DER*. También se modifica la función *evaluar* de aplicación de un árbol al espacio de juego para que tenga en cuenta los nuevos conjuntos de operandos y operadores. En el cálculo de la adaptación se consideran las distancias de todos los alienígenas que están en la misma columna que el defensor. Al calcular estas distancias hay que tener en cuenta la dirección del movimiento del alienígena, ya que si se están alejando no se encontrarán hasta que el alien llegue al extremo del tablero y dé la vuelta.

En la comprobación de la *Terminacion* del juego, ahora hay que tener en cuenta que el jugador no gana hasta haber destruido todos los alienígenas del espacio.

En estas versiones más complejas, en las que muchas ejecuciones no consiguen llegar a encontrar una buena estrategia, se deben analizar los mejores programas obtenidos en una secuencia de ejecuciones tan amplia como sea posible.

Se pueden probar otras versiones aún más complejas en las que se introduzcan nuevos elementos, como permitir refugios para el defensor.

Apéndice A

UTILIDADES: CÓDIGO JAVA

A continuación se incluyen fragmentos de código Java con algunas operaciones útiles de propósito general que se pueden aplicar a los proyectos vistos anteriormente. También se incluye un ejemplo de interfaz gráfica para la aplicación de optimización de funciones del Proyecto 1 y gráficas con los resultados de la ejecución del algoritmo.

Operaciones algoritmo genético simple

Selección por ruleta

```
public void seleccionRuleta() {
 // Seleccionados para sobrevivir
 int[] sel_super = new int[tamaño_pob];
 double prob;
 int pos_super;
 for(int i = 0; i < tamaño_pob; i++) {
       prob = Math.random();
       pos_super = 0;
        while((prob > poblacion[pos_super].getPuntAcum()) &&
                                (pos_super < tamaño_pob))
                                              pos_super++;
       sel_super[i] = pos_super;
       }
```

```
 // Se genera la población intermedia
 Cromosoma[] nuevaPob = new Cromosoma[tamaño_pob];
 for(int i = 0; i < tamaño_pob; i++) {
       nuevaPob[i].copia(poblacion[sel_super[i]]);
       }
 poblacion = nuevaPob;
}
```

Método reproducción

```
public void reproduccion(){
 //seleccionados para reproducir
 int[] sel_cruce = new int[tam_pob];
 //contador seleccionados
 int num_sele_cruce = 0;
 double prob;
 Cromosoma hijo1, hijo2;
 hijo1 = creaCromosoma();
 hijo2 = creaCromosoma();

 //se eligen los individuos a cruzar
 for(int i = 0; i < tam_pob; i++){
  //se generan tam_pob números aleatorios en [0,1)
  prob = (float)Math.random();
  if(prob < prob_cruce){
             sel_cruce[num_sele_cruce] = i;
             num_sele_cruce++;
             }
 }

  //el número de seleccionados se hace par
  if((num_sele_cruce % 2) == 1)
             num_sele_cruce--;
  for(int i = 0; i < num_sele_cruce; i+=2){
    cruce(pob[sel_cruce[i]],pob[sel_cruce[i+1]], hijo1, hijo2);

   //los nuevos individuos sustituyen a sus progenitores
   pob[sel_cruce[i]].setGenes(hijo1.getGenes());
   pob[sel_cruce[i+1]].setGenes(hijo2.getGenes());
  }
}
```

Método cruce

```
public void cruce(Cromosoma padre1, Cromosoma padre2,
                  Cromosoma hijo1,Cromosoma hijo2,int punto_cruce) {

   // Cruce monopunto
   // Primera parte del intercambio: 1 a 1 y 2 a 2
   for (int i = 0; i < punto_cruce; i++){
                    hijo1.getGenes()[i] = padre1.getGenes()[i];
                    hijo2.getGenes()[i] = padre2.getGenes()[i];
              }
   // Segunda parte del intercambio: 1 a 2 y 2 a 1
   for (int i = punto_cruce; i < padre1.getLongitud(); i++){
                    hijo1.getGenes()[i] = padre2.getGenes()[i];
                    hijo2.getGenes()[i] = padre1.getGenes()[i];
              }
   // Se evalúan
   hijo1.evalua();
   hijo2.evalua();
}
```

Método mutación

```
public void mutacion() {
 // mutación a nivel de bit
 boolean mutado;
 double prob;
 for(int i = 0; i < tamaño_pob; i++) {
  mutado = false;
  for(int j = 0; j < poblacion[i].getLongitud(); j++) {
       prob = Math.random();
       if(prob < prob_mut) {
             poblacion[i].getGenes()[j]=!poblacion[i].getGenes()[j];
             mutado = true;
       }
      }
   if(mutado)
      //si muta se recalcula la aptitud
      poblacion[i].setAptitud(poblacion[i].evalua());
   }
}
```

Método evaluar población

```
public void evaluarPoblacion() {
 maximo = Integer.MIN_VALUE;
 double aux = 0;
 double sumaAptitud = 0;
 double punt_acu = 0;

//Búsqueda del máximo
 for(int i = 0; i < tamaño_pob; i++) {
       aux = poblacion[i].evalua();
       sumaAptitud += aux;
       if(aux > maximo) {
              maximo = aux;
              pos_mejor = i;
              }
  }

for(int i = 0; i < tamaño_pob; i++) {
 poblacion[i].setPuntuacion(poblacion[i].getAptitud()/sumaAptitud);
 poblacion[i].setPunt_acu(poblacion[i].getPuntuacion() + punt_acu);
 punt_acu += poblacion[i].getPuntuacion();
}

if(!maximizar) {//Si es un problema de minimización
       sumaAptitud = 0;
       for(int i = 0; i < tamaño_pob; i++) {
              aux = maximo - poblacion[i].evalua();
              poblacion[i].setAptitud(aux);
              sumaAptitud += aux;
              }
       for(int i = 0; i < tamaño_pob; i++) {
              if(poblacion[i].getAptitud() > maximo) {
                     maximo = poblacion[i].getAptitud();
                     pos_mejor = i;
                     }
              }
 punt_acu = 0;
 for(int i = 0; i < tamaño_pob; i++) {
  poblacion[i].setPuntuacion(poblacion[i].getAptitud()/sumaAptitud);
  poblacion[i].setPunt_acu(poblacion[i].getPuntuacion() + punt_acu);
  punt_acu += poblacion[i].getPuntuacion();
  }
}
// Si el mejor de esta generación es mejor que el
// mejor que había antes, se actualiza
if(maximo > elMejor.getAptitud())
              elMejor.copia(poblacion[pos_mejor]);
}
```

Ejemplo Interfaz gráfica

A continuación se muestra un ejemplo de interfaz gráfica sencilla pero muy representativa de este tipo de problemas. Podemos seleccionar los diferentes parámetros para el algoritmo: tamaño de la población, número de generaciones, probabilidad de cruce, probabilidad de mutación y otras variables. En este caso se muestra un ejemplo de aplicación para el proyecto de optimización de funciones.

Se deja a elección del lector el código correspondiente a la interfaz gráfica y a la generación de gráficas y sólo se especifica la parte correspondiente al código del algoritmo genético.

El esquema propuesto para la ventana principal es el siguiente:

```
. . . .
class VentanaPrincipal extends JFrame {

// Definir y configurar componentes de la interfaz
// Obtener los valores de los parámetros de la interfaz gráfica
// . . .

public VentanaPrincipal(){

// Creamos un algoritmo genético con los valores obtenidos
// de la interfaz
```

```
AGenetico AG = new AGenetico(numGeneraciones, tamPoblacion,
                             pCruce, pMutacion, precision,
                             idFuncion, numParam);
AG.inicializa();
AG.evaluarPoblacion();
while (!AG.terminado()){
       AG.num_gen++;
       if (elitismo())  AG.extraerElite();
       AG.seleccion();
       AG.reproduccion();
       AG.mutacion();
       if (elitismo()) AG.insertarElite();
       AG.evaluarPoblacion();
```

```
//Dibujo de gráficas y resultados
//...

 public static void main(String[] args)   {
     VentanaPrincipal v = new VentanaPrincipal();
 }
}
```

A continuación se muestran algunos ejemplos de ejecuciones y los resultados obtenidos, junto con la gráfica de evolución hacia los máximos o los mínimos. Aunque se podrían incluir muchas gráficas con diferentes valores de parámetros, hemos seleccionado un ejemplo representativo de ejecución para cada función.

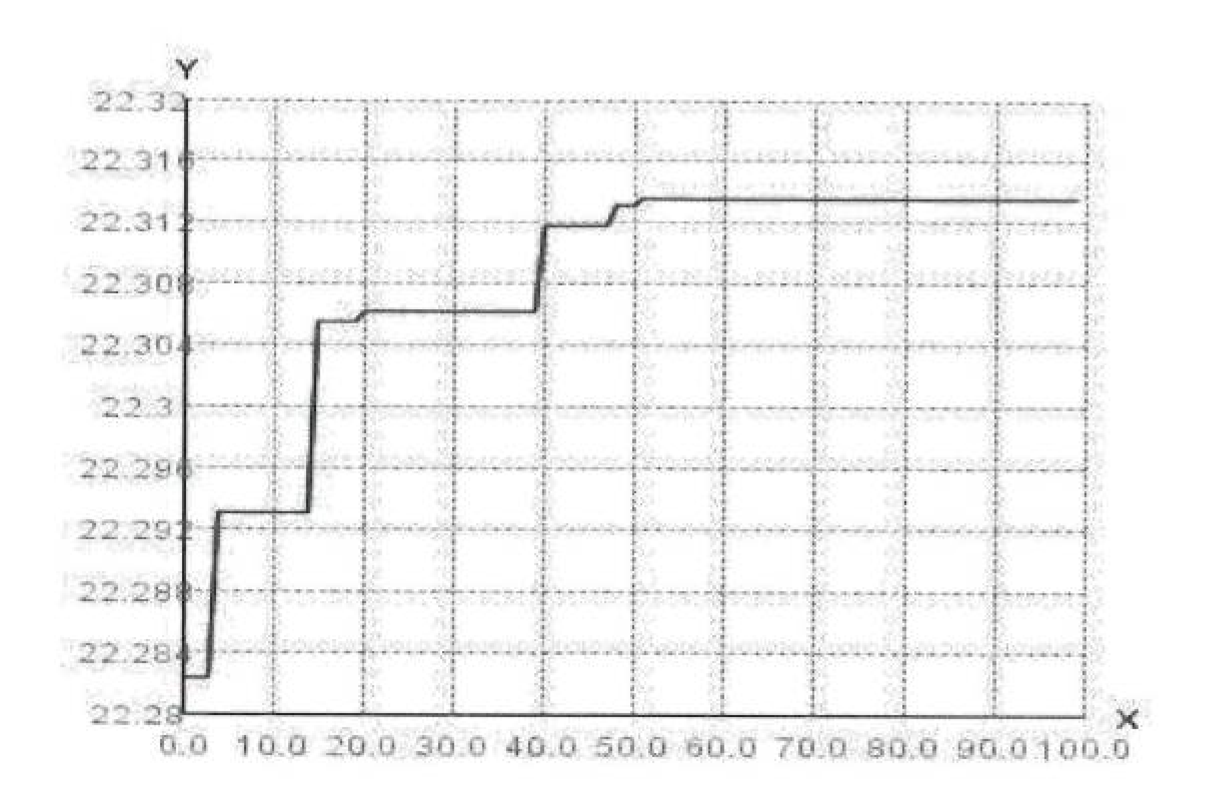
Ejemplo gráficas Proyecto 1

Función 1

- $f(x) = 20 + e - 20e^{-0.2|x|} - e^{\cos 2\pi x} : x \in [0, 32]$

Parámetros:

Tamaño población: 50
Número máximo de generaciones: 100
Probabilidad de cruce: 0.6
Probabilidad de mutación: 0.1



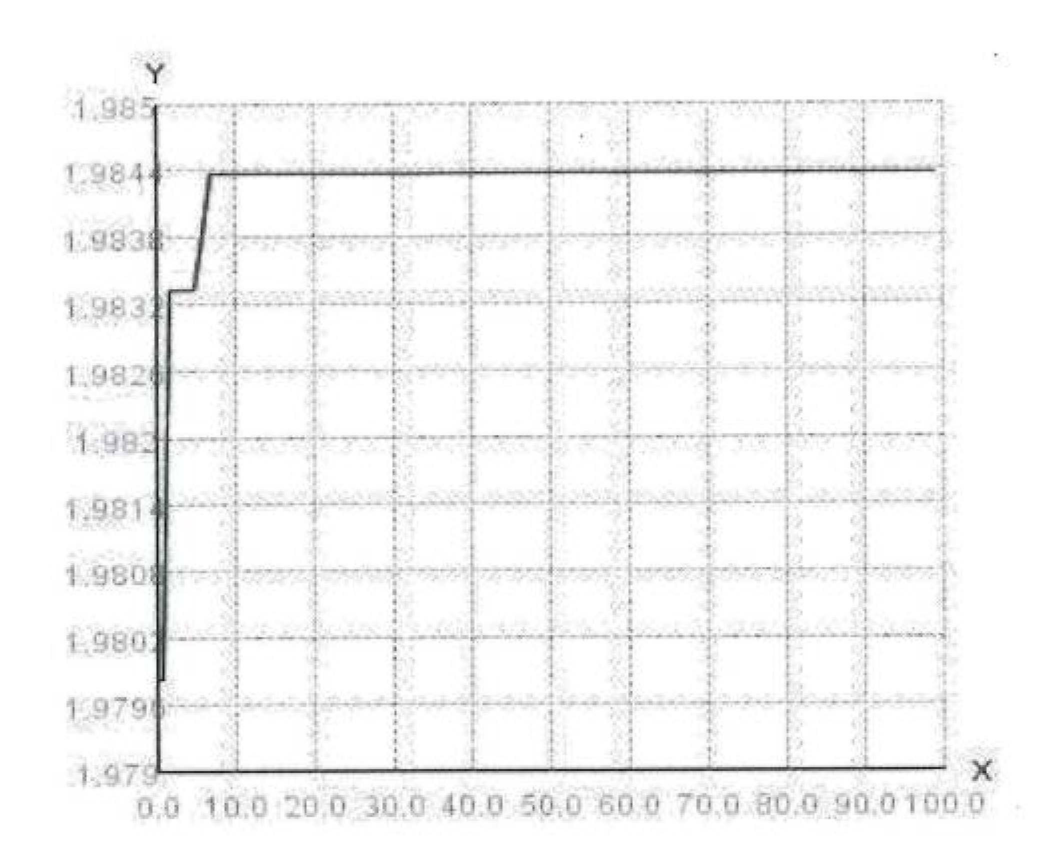

```
x: 31.500853540141183
f(x) = 22.313677270291098
```

Función 2

- $f(x) = x + |\,sen(32\pi x)\,| \; : \; x \in [0,1]$

Parámetros:

Tamaño población: 50
Número máximo de generaciones: 100
Probabilidad de cruce: 0.6
Probabilidad de mutación: 0.1



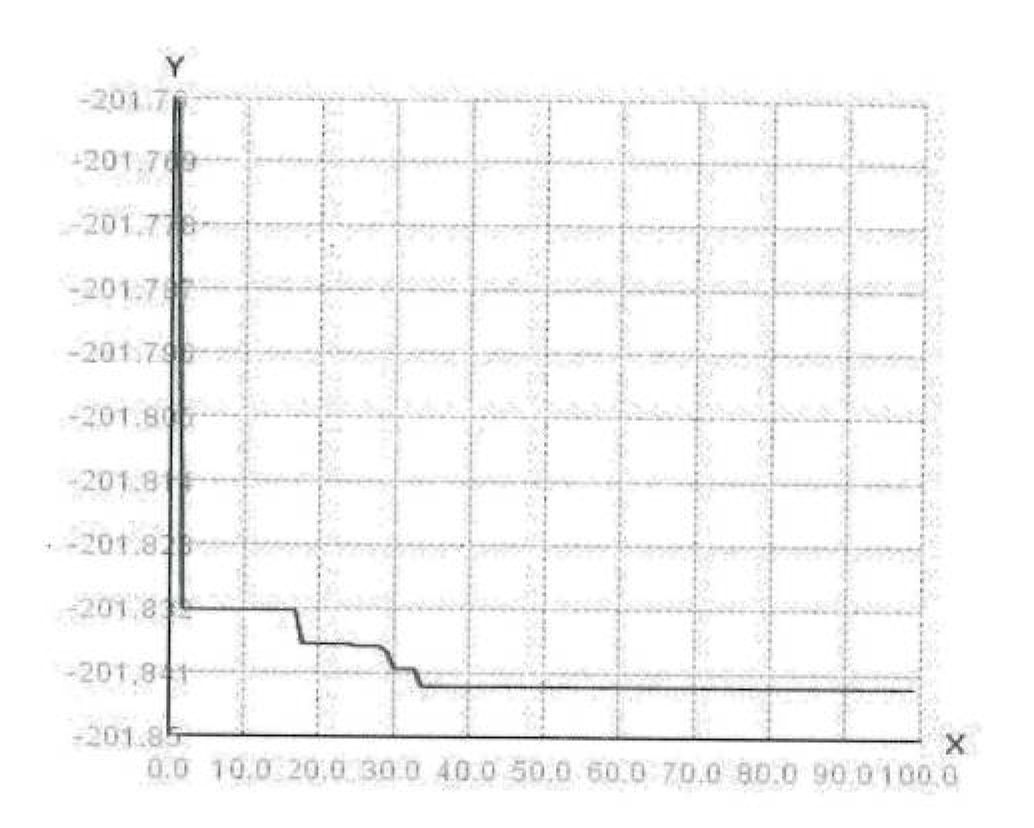

```
x: 0.9843740462674724
f(x)= 1.9843740416710185
```

Función 3

- $f(x) = -|\, xsen(\sqrt{|\, x\,|})\,| : x \in [-250{,}250]$

Parámetros:

Tamaño población: 50
Número máximo de generaciones: 100
Probabilidad de cruce: 0.6
Probabilidad de mutación: 0.1



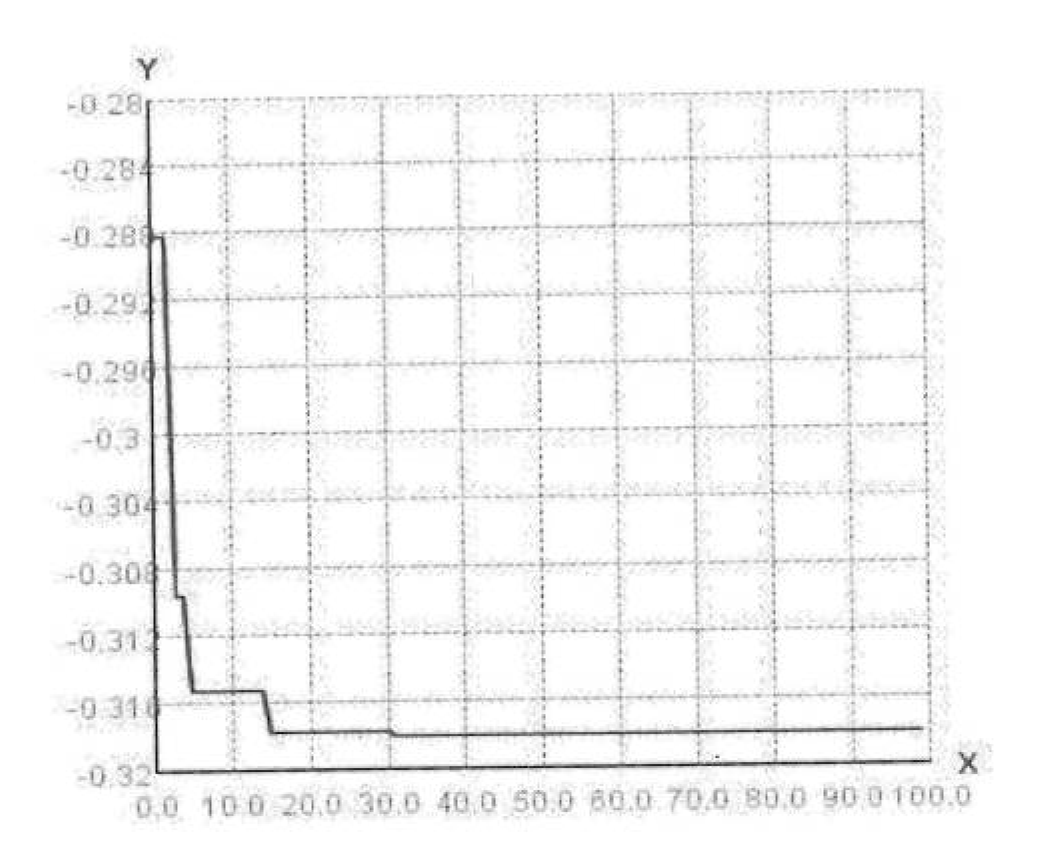

```
x: -203.81134197847152
f(x) = -201.84321680223027
```

Función 4

- $f(x) = \dfrac{senx}{1+\sqrt{x}+\dfrac{\cos x}{1+x}} : x \in [0, 25]$

Parámetros:

Tamaño población: 50
Número máximo de generaciones: 100
Probabilidad de cruce: 0.6
Probabilidad de mutación: 0.1



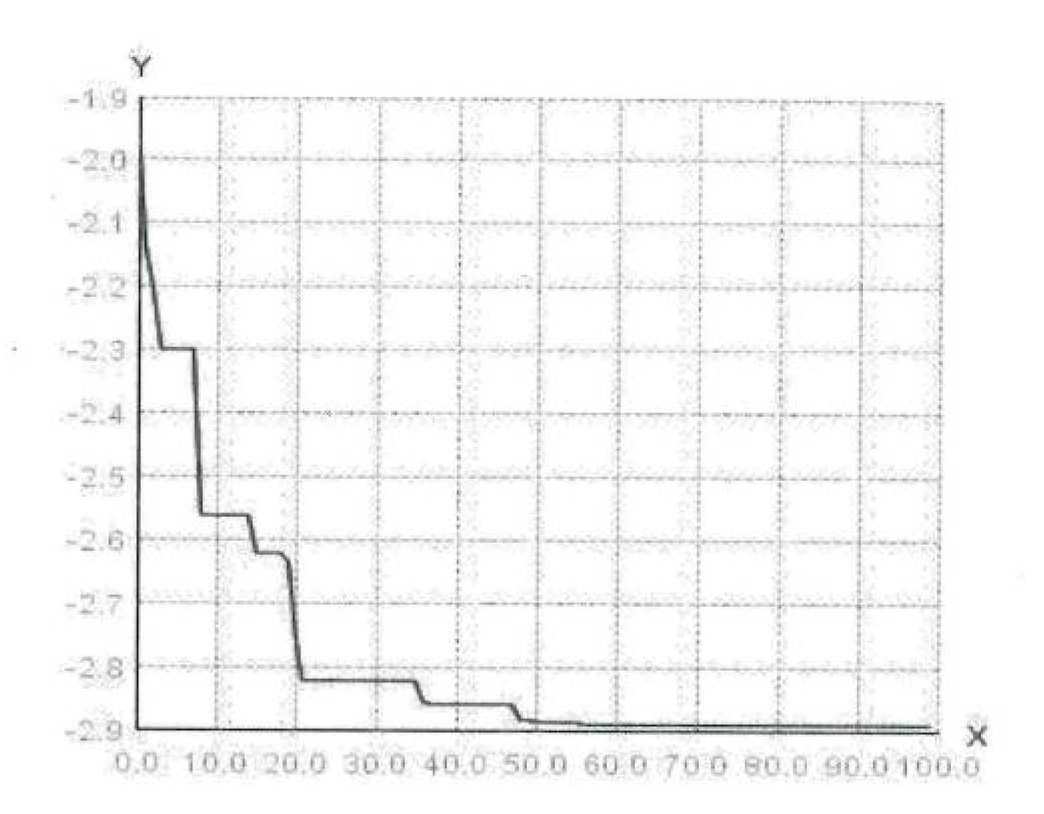

```
x: 4.581278157341604
f(x) = -0.3180710368195209
```

Función 5

- $$f(x_i \mid i = 1..n) = -\sum_{i=1}^{n} sen(x_i) sen^{20}\left(\frac{(i+1)x_i^{2}}{\pi}\right) : x_i \in [0,\pi]$$

Parámetros:

Tamaño población: 50
Número máximo de generaciones: 100
Probabilidad de cruce: 0.6
Probabilidad de mutación: 0.1



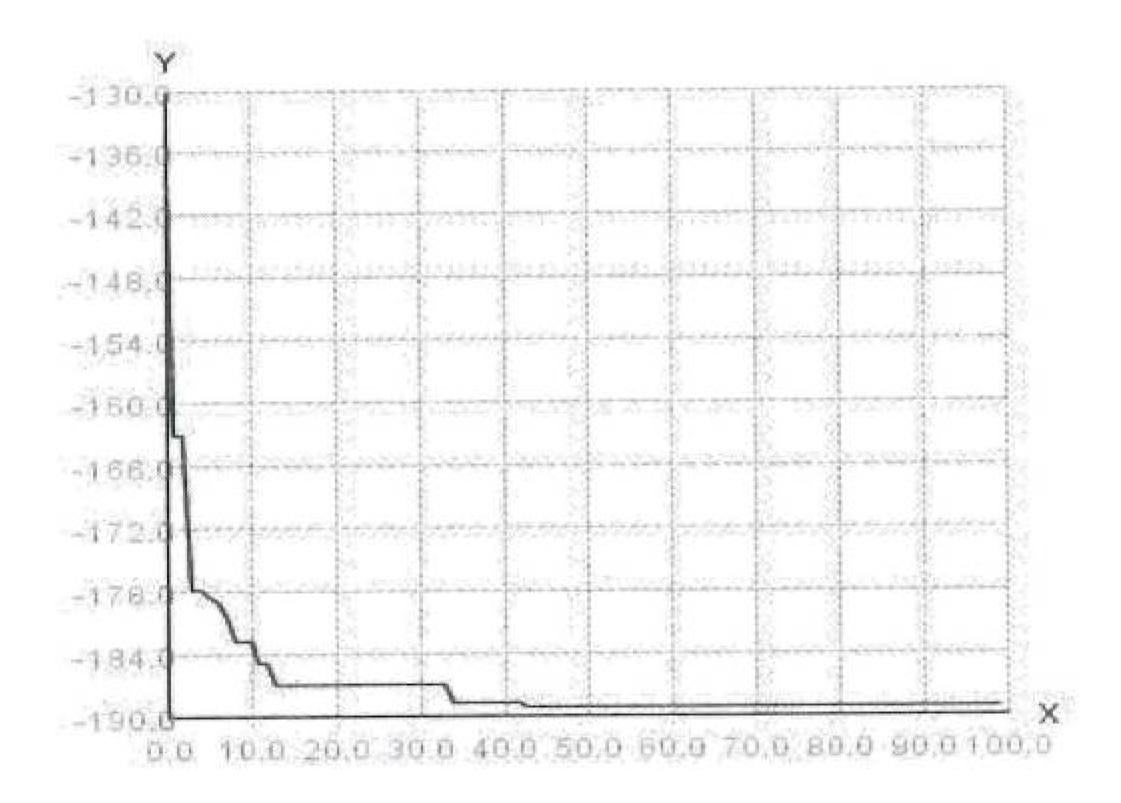

```
X1: 1.5403554665539603
X2: 1.2745851843874236      f(x) = -2.823400257879652
X3: 1.9349840673466965
```

Función 6

- $f(x_i \mid i=1..n) = \sum_{i=1}^{n} -x_i sen(\sqrt{|x_i|}) : x_i \in [0,100]$

Parámetros:

Tamaño población: 50
Número máximo de generaciones: 100
Probabilidad de cruce: 0.6
Probabilidad de mutación: 0.1




```
X1: 65.68676537205255
X2: 65.2346279474525
f(x) = -127.2544199667835
```

Representación con permutaciones: operaciones básicas en Java

A continuación se incluye el código Java correspondientes a algunos operadores utilizados en algoritmos evolutivos con representación basada en permutaciones. Hemos seleccionado los métodos más comunes.

Cruce por orden (OX)

El cruce por orden consiste en copiar en cada uno de los hijos una subcadena de uno de los padres mientras se mantiene el orden relativo de las ciudades que aparecen en el otro padre.

```
public class CruceOrden {

void cruce(Cromosoma padre1, Cromosoma padre2) {

  // puntos de cruce p1 y p2
  int p1 = (int)(Math.random() * (Cromosoma.longitudCromosoma + 1));
  int p2 = (int)(Math.random() * (Cromosoma.longitudCromosoma + 1));

  if (p1 == p2) return;

  if (p2 < p1) {
            int tmp = p1;
            p1 = p2;
            p2 = tmp;
      }
  if(p1 == 0 && p2 == Cromosoma.lonCrom)
                return;
  int[] genes1 = trozos(padre1, padre2, p1, p2);
  int[] genes2 = trozos(padre2, padre1, p1, p2);

  padre1.genes = genes1;
  padre2.genes = genes2;
 }


 int[] trozos(Cromosoma padre1,Cromosoma padre2,int p1,int p2) {

      int[] genes = new int[Cromosoma.lonCrom];
      int i; // posicion de inserción
      for (i = p1; i < p2; i++)
                   genes[i] = padre2.genes[i];
      int pos = p2 % Cromosoma.lonCrom;
```

```
 for (i %=Cromosoma.lCrom;i!= p1;i=(i+1)%Cromosoma.lCrom) {
   while (Cromosoma.indexOf(genes, padre1.genes[pos]) != -1) {
             pos = ((pos + 1) % Cromosoma.lCrom);
             }
   genes[i] = padre1.genes[pos];
   pos = ((pos + 1) % Cromosoma.lCrom);
  }

  return genes;
 }
}
```

Cruce por Emparejamiento Parcial (PMX)

El cruce por emparejamiento parcial consiste en elegir un tramo de la ruta de uno de los padres y cruzar manteniendo el orden y la posición de la mayor cantidad posible de ciudades del otro.

```
public class CrucePMX {

void cruce(Cromosoma padre1, Cromosoma padre2) {

// puntos de cruce p1 y p2
  int p1 = (int)(Math.random() * (Cromosoma.lonCrom + 1));
  int p2 = (int)(Math.random() * (Cromosoma.lonCrom + 1));

  if (p1 == p2) return;

  if (p2 < p1) {
             int tmp = p1;
             p1 = p2;
             p2 = tmp;
      }
  if(p1 == 0 && p2 == Cromosoma.lonCrom)
                return;
  int[] genes1 = trozos(padre1, padre2, p1, p2);
  int[] genes2 = trozos(padre2, padre1, p1, p2);

  padre1.genes = genes1;
  padre2.genes = genes2;
 }

int[] trozos(Cromosoma padre1, Cromosoma padre2, int p1, int p2) {
 int[] genes = new int[Cromosoma.lonCrom];
 for (int i = 0; i < Cromosoma.lonCrom; i++) {
   if (p1 <= i && i < p2) {
```

```
                genes[i] = padre2.genes[i];
            }
    else {
            int valor = padre1.genes[i];
            int pos_en_otro_crom = p1;
             do {
                 if (padre2.genes[pos_en_otro_crom] == valor) {
                     valor = padre1.genes[pos_en_otro_crom];
                     pos_en_otro_crom = p1;
                     continue;
                     }
                 pos_en_otro_crom++;
                } while (pos_en_otro_crom < p2);
               genes[i] = valor;
               }
        }
     return genes
   }
}
```

Mutación por inversión

En este tipo de mutación se altera el orden de una subcadena o tramo del individuo. Se aplica con una determinada probabilidad y consiste en seleccionar dos puntos del individuo al azar e invertir los elementos que hay entre dichos puntos.

```
public class MutacionInversion {


public void mutar(Cromosoma individuo){

//Seleccionamos los puntos al azar
  int p1 = (int)(Math.random() * (individuo.lonCrom + 1));
  int p2 = (int)(Math.random() * (individuo.lonCrom + 1));

 while(p1 == p2){
        p1 = (int)(Math.random()*(individuo.lonCrom));
        p2 = (int)(Math.random()*(individuo.lonCrom));
 }

 if(p1 > p2){
              int aux = p1;
              p1 = p2;
              p2 = aux;
 }

for (int i = 0; i < (int)((p2 - p1 + 1)/2); i++) {
```

```
  intercambiaGenes(individuo.genes, p1 + i, p2 - i);
  }
}

void intercambiaGenes(int[] genes, int p1, int p2) {
        int tmp = genes[p1];
        genes[p1] = genes[p2];
        genes[p2] = tmp;
 }
}
```

Mutación por Inserción

En este tipo de mutación se inserta una o varias ciudades elegidas al azar en unas posiciones también elegidas al azar. Puede haber una o varias inserciones.

```
public class MutacionInsercion {

// Se inserta uno o varios elementos elegidos al azar en
// posiciones elegidas al azar.

public void mutar(Cromosoma individuo){

//Seleccionamos un elemento y una posición al azar
 int elemento = (int)(Math.random()*(individuo.lonCrom));
 int posicion = (int)(Math.random()*(individuo.lonCrom -1));
 int i = -1;

 boolean encontrado = false;
 while(not encontrado){
       i++;
       if(individuo.genes[i] == elemento)
                       encontrado = true;
       }
 if(i > posicion){
       for(int j = i; j > posicion; j--){
                    individuo.genes[j] = individuo.genes[j-1];
        }
     }
   else{
        for(int j = i; j < posicion; j++){
                    individuo.genes[j] = individuo.genes[j+1];
              }
        }
       individuo.genes[posicion] = elemento;
  }
 }
```

Mutación heurística

En este tipo de mutación se seleccionan n elementos del individuo al azar y se generan todas las permutaciones con los elementos seleccionados, obteniendo como resultado de la mutación el de mejor adaptación:

```
public class MutacionHeuristica {

// Seleccionamos por ejemplo 3 elementos al azar y generamos
// las permutaciones de dichos elementos, seleccionando
// el mejor de todos los individuos obtenidos

public void mutar(Cromosma individuo){

 int[] v =  new int[3]; //valores elegidos
 int[][] permuta = new int[6][]; //permutaciones

 v[0] = (int)Math.floor(Math.random()*individuo.lonCrom);
 do{
     v[1] = (int)Math.floor(Math.random()*individuo.lonCrom);
   }while (v[1] == v[0]);
 do{
     v[2] = (int)Math.floor(Math.random()*individuo.lonCrom);
 }while ((v[2] == v[0])||(v[2] == v[1]));

 //Generamos los 6 posibles individuos con las permutaciones
 permuta[0] = new int[]{v[0],v[1],v[2]};
 permuta[1] = new int[]{v[0],v[2],v[1]};
 permuta[2] = new int[]{v[1],v[0],v[2]};
 permuta[3] = new int[]{v[1],v[2],v[0]};
 permuta[4] = new int[]{v[2],v[0],v[1]};
 permuta[5] = new int[]{v[2],v[1],v[0]};
 Cromosoma auxiliar = new Cromosoma();
 for (int i = 0;i < individuo.lonCrom; i++){
      auxiliar.genes[i] = individuo.genes[i];
}

 //Selección del mejor de los obtenidos
 int ind = 0;
 float mejorValor = 100000;
 for (int i = 0;i < permuta.length;i++){
      auxiliar.genes[permuta[i][0]] = individuo.genes[v[0]];
      auxiliar.genes[permuta[i][1]] = individuo.genes[v[1]];
      auxiliar.genes[permuta[i][2]] = individuo.genes[v[2]];
      float valor = (float)auxiliar.evalua();
       if (valor < mejorValor){
             mejorValor = valor;
             ind = i;
             }
      }
```

```
 auxiliar.genes[permuta[mejorCromosoma][0]] = individuo.genes[v[0]];
 auxiliar.genes[permuta[mejorCromosoma][1]] = individuo.genes[v[1]];
 auxiliar.genes[permuta[mejorCromosoma][2]] = individuo.genes[v[2]];

 for (int i = 0;i < individuo.lonCrom;i++){
             individuo.genes[i] = auxiliar.genes[i];;
       }
  }
}
```

Apéndice B

SOLUCIÓN PROYECTO 1: CÓDIGO C++

A continuación se incluye el código C++ correspondiente al Proyecto 1. Hemos seleccionado una función de cada uno de los tipos propuestos: maximización, minimización y multivarible.

Se utiliza programación estructurada y clases predefinidas de C++ como la clase **vector**.

Los módulos que se presentan son los siguientes:

- `funciones.h`: módulo con las definiciones de tipos y prototipos de funciones.
- `Funciones.cpp`: módulo con la implementación de las funciones.

La función a optimizar se especifica en la función adaptación.

El programa se puede invocar desde la línea de comandos, especificando los siguientes parámetros:

- Límite inferior del intervalo en que busca el óptimo
- Límite superior
- Tolerancia o exactitud con la que se pide el resultado
- Tamaño de población
- Número de generaciones de evolución

- Tasa de cruce
- Tasa de mutación

Por ejemplo, para la primera función que se presenta:

```
funciones 0.0 32.0 0.0001 500 1000 0.4 0.001
```

Se describen también los cambios que se introducen para la minización de una función y para trabajar con funciones de varias variables.

Módulo funciones.h

```
#include <stdlib.h>
#include <math.h>
#include <string>
#include <vector.h>

using namespace std;
// individuo
typedef vector < bool > TGenes;
typedef struct{
          TGenes genes;         // cadena de bits (genotipo)
          double x;             // fenotipo
          double aptitud;       // funcion de evaluacion
          double puntuacion;    // punt. relativa:fitness/sumfitness
          double punt_acu;      // puntuacion acumulada para sorteos
          bool elite;           // indicativo de pertenecer a la elite
        }TIndividuo;
// población
typedef vector <TIndividuo> TPoblacion;


/*************************/
/* Funciones del módulo */
/*************************/
TPoblacion poblacion_inicial(int tam_pob, int lcrom, double x_min,
                              double x_max);
TIndividuo genera_indiv(int lcrom, double x_min, double x_max);
void seleccion(TPoblacion& pob, int tam_pob);
double adaptacion(TIndividuo& individuo, int lcrom, double x_min,
                   double x_max);
double decod(TGenes genes, int lcrom, double x_min, double x_max);
int bin_dec(TGenes genes, int lcrom);

void reproduccion(TPoblacion& pob, int tam_pob, int lcrom,
              double prob_cruce, double x_min, double x_max);
void cruce(TIndividuo padre1, TIndividuo padre2, TIndividuo& hijo1,
```

```
            TIndividuo& hijo2,int lcrom, int punto_cruce,
            double x_min, double x_max);
void mutacion(TPoblacion& pob, int tam_pob, int lcrom,
              double prob_mut,double x_min, double x_max);
void evaluacion(TPoblacion& pob, int tam_pob, int& pos_mejor,
                double& sumfitness);
```

Módulo funciones.cpp

Representa la función:

- $f(x) = 20 + e - 20e^{-0.2|x|} - e^{\cos 2\pi x} : x \in [0, 32]$

que presenta un máximo (22.3136) en 31.5005.

```
/*************************************************************************
/* Programacion evolutiva: proyecto 1                                   */
/* version 1.0:                                                         */
/*                                                                      */
/* Modulo funciones.cpp                                                 */
/*                                                                      */
/*************************************************************************

#include "funciones.h"

int main( int argc, char *argv[] )
{
  double x_min;            // límite inferior del intervalo
  double x_max;            // límite superior del intervalo
  double TOL;              // Tolerancia
  int tam_pob;             // tamaño de población
  int num_gen;             // número de generaciones
  double prob_cruce;       // probabilidad de cruce
  double prob_mut;         // probabilidad de mutación
  TPoblacion pob;          // población
  int lcrom;               // longitud de los individuos
  int generacion;          // contador de generaciones
  int pos_mejor;           // posición del mejor individuo
  double sumaptitud;       // suma de adaptaciones

  sscanf(argv[1], "%lf", &x_min ); // límite inferior del intervalo
                                   // en que se busca el óptimo
  sscanf(argv[2], "%lf", &x_max ); // límite superior del intervalo
                                   // en que se busca el óptimo
  sscanf(argv[3], "%lf", &TOL ); // tolerancia o exactitud
                                 // con que se busca el óptimo
```

```
  sscanf(argv[4], "%d", &tam_pob ); // tamaño de la población
  sscanf(argv[5], "%d", &num_gen ); // número de iteraciones
  sscanf(argv[6], "%lf", &prob_cruce ); // porcentaje de cruces
  sscanf(argv[7], "%lf", &prob_mut );  // porcentaje de mutaciones

  // inicializaciones
  lcrom = (int)ceil(log(1 + (x_max - x_min)/TOL) / log((double)2));

  // inicialización de números aleatorios
  time_t t;
  srand((unsigned) time(&t));

  // se genera la población inicial
  pob = poblacion_inicial(tam_pob, lcrom, x_min, x_max);
  // se evalúa la población
  evaluacion(pob, tam_pob, pos_mejor, sumaptitud);

  // bucle de evolución
  for (generacion=0; generacion < num_gen; generacion++){
    seleccion(pob, tam_pob);
    reproduccion(pob, tam_pob, lcrom, prob_cruce, x_min, x_max);
    mutacion(pob, tam_pob, lcrom, prob_mut, x_min, x_max);
    evaluacion(pob, tam_pob, pos_mejor, sumaptitud);
  }
  cout << "Aptitud del mejor(" << pos_mejor << ") = ";
  cout <<  pob[pos_mejor].aptitud << " en " << pob[pos_mejor].x <<
endl;
}
/*****************************************************************/
/*                                                               */
/* funcion: poblacion_inicial                                    */
/* Genera la poblacion inicial aleatoriamente                    */
/*                                                               */
/*****************************************************************/
TPoblacion poblacion_inicial(int tam_pob, int lcrom,
                             double x_min, double x_max)
{
  TPoblacion pob;
  TIndividuo indiv;
  int i;
  for(i=0; i < tam_pob; i++){
    indiv = genera_indiv(lcrom, x_min, x_max);
    pob.push_back(indiv);
  }
  return pob;
}
/*****************************************************************/
/*                                                               */
/* funcion: genera_indiv                                         */
/* Genera un individuo de la poblacion inicial                   */
/*                                                               */
/*****************************************************************/
```

```
TIndividuo genera_indiv(int lcrom, double x_min, double x_max)
{
  int i;
  TIndividuo indiv;
  int gen;

  indiv.genes.clear();
  for(i=0; i < lcrom; i++){
    if (rand() < RAND_MAX / 2 )
      gen = 0;
    else
      gen = 1;
    indiv.genes.push_back(gen);
  }
    indiv.aptitud = adaptacion(indiv, lcrom,. x_min, x_max);
  return indiv;
}


/******************************************************************/
/*                                                                */
/* funcion: seleccion                                             */
/* Selecciona un unico individuo por el metodo de la ruleta      */
/*                                                                */
/******************************************************************/

void seleccion(TPoblacion& pob, int tam_pob)
{
  int* sel_super;                  // seleccionados para sobrevivir
  float prob,f;                    // probabilidad de selección
  int pos_super;                   // posición del superviviente
  TPoblacion pob_aux;              // población auxiliar
  int i,j;
  int TAM_ELITE = pob.size()* 2 /100;
  int* sel_elite;

  sel_super = new int[tam_pob];
  sel_elite = new int[TAM_ELITE];
  //------------------------//
  // se selecciona la elite
  //------------------------//
  for (i=0; i < TAM_ELITE; i++){
    sel_elite[i] = 0;
  }
  for (i=0; i < tam_pob; i++){
    pob[i].elite = false;
    j = 0;
    while ( (j < TAM_ELITE) &&
        (pob[sel_elite[j]].aptitud >= pob[i].aptitud)){
      j++;
    }
    if (j < TAM_ELITE){
```

```
      sel_elite[j] = i;
    }
  }
  for (i=0; i < TAM_ELITE; i++){
    pob[sel_elite[i]].elite = true;
  }

  // se seleccionan tam_pob individuos para reproducirse
  // se generan números aleatorios entre 0 y 1,
  // seleccionan individuos de acuerdo con su puntuación acumulada

  for (i=0; i < tam_pob; i++){
    if (pob[i].elite == true)  //ELITISMO
      sel_super[i] = i;
    else{
      f = rand();
      prob = (f/(RAND_MAX+1.0));
      pos_super = 0;
      while ((prob>pob[pos_super].punt_acu)&&(pos_super < tam_pob))
        pos_super++;
      if (pos_super < tam_pob)
        sel_super[i] = pos_super;
      else
        sel_super[i] = pos_super-1;
    }
  }
  // se genera la población intermedia
  for (i=0; i < tam_pob; i++){
    pob_aux.push_back(pob[sel_super[i]]);
  }
  pob.clear();
  for (i=0; i < tam_pob; i++){
    pob.push_back(pob_aux[i]);
  }
  delete [] sel_super;
  delete [] sel_elite;
}

/*****************************************************************/
/*                                                               */
/* funcion: adaptacion                                           */
/* Evalua la calidad de un individuo                             */
/*                                                               */
/*****************************************************************/


double adaptacion(TIndividuo& individuo, int lcrom,
                  double x_min, double x_max)
{
  double x; // fenotipo
  double f; // valor de la función a optimizar
```

```
  x = decod(individuo.genes, lcrom, x_min, x_max);

  individuo.x = x;

  // AQUÍ se especifica la función a optimizar
  f = 20+exp((double)1)-20*exp((double)(-0.2*fabs(x)))-
         exp((double)(cos(2*M_PI*x)));

  return f;
}


/****************************************************************/
/*                                                              */
/* funcion: decod                                               */
/* Decodifica el genotipo de un individuo                       */
/*                                                              */
/****************************************************************/

double decod(TGenes genes, int lcrom, double x_min, double x_max)
{
  double x;

  x=(float)bin_dec(genes, lcrom)/(pow((double)2,(double)lcrom) - 1);
  x = x_min + (x_max - x_min) * x;
  return x;
}

/****************************************************************/
/*                                                              */
/* funcion: bin_dec                                             */
/* Convierte de binario a decimal                               */
/*                                                              */
/****************************************************************/

int bin_dec(TGenes genes, int lcrom)
{
  int i, d=0, pot=1;

  for(i=0; i < lcrom; i++){
    d = d + pot*genes[i];
    pot = pot * 2;
  }
  return d;
}
/****************************************************************/
/*                                                              */
/* funcion: reproduccion                                        */
/* Selecciona los individuos a reproducirse y aplica el         */
/* operador de cruce                                            */
/*                                                              */
/****************************************************************/
```

```
void reproduccion(TPoblacion& pob, int tam_pob, int lcrom,
           double prob_cruce, double x_min, double x_max)
{
  int* sel_cruce;       //seleccionados para reproducirse
  int num_sel_cruce=0; //num de individuos seleccionados a cruzar
  double prob;
  int punto_cruce;
  TIndividuo hijo1, hijo2;
  int i;

  sel_cruce = new int[tam_pob];
  // se eligen los individuos a cruzar
  for (i=0; i < tam_pob; i++){
    // se generan tam_pob numeros aleatorios a_i en [0 1)
    prob = (rand()/(RAND_MAX+1.0));
    // se eligen para el cruce los individuos de
    // las posiciones i con a_i < prob_cruce
    if (prob < prob_cruce){
      sel_cruce[num_sel_cruce] = i;
      num_sel_cruce++;
    }
  }
  // el número de selecionados se hace par
  if ((num_sel_cruce % 2) == 1)
    num_sel_cruce--;

  // se cruzan los individuos elegidos en un punto al azar
  punto_cruce = (int)(rand() * (lcrom+1) / (RAND_MAX + 1.0)) + 1;
  for (i=0; i < num_sel_cruce; i=i+2){
    cruce(pob[sel_cruce[i]], pob[sel_cruce[i+1]], hijo1, hijo2,
          lcrom, punto_cruce, x_min, x_max);
    // los nuevos individuos sustituyen a sus progenitores
    // respetando la élite
   if ((hijo1.aptitud<adaptacion(pob[sel_cruce[i]],lcrom,x_min,
x_max)) || (!pob[sel_cruce[i]].elite)){ //ELITISMO
      pob[sel_cruce[i]] = hijo1;
    }
    if ((hijo2.aptitud <
            adaptacion(pob[sel_cruce[i+1]], lcrom, x_min, x_max))
        || (!pob[sel_cruce[i+1]].elite)){
      pob[sel_cruce[i+1]] = hijo2;
    }
  }
  delete [] sel_cruce;
}
/***************************************************************/
/*                                                             */
/* funcion: cruce                                              */
/* Aplica el operador de cruce                                 */
/*                                                             */
/***************************************************************/
```

```
void cruce(TIndividuo padre1, TIndividuo padre2, TIndividuo& hijo1,
TIndividuo& hijo2, int lcrom, int punto_cruce, double x_min, double
x_max)
{
  int i;
  hijo1.genes.clear();
  hijo2.genes.clear();

  // primera parte del intercambio: 1 a 1 y 2 a 2
  for(i=0; i < punto_cruce; i++){
    hijo1.genes.push_back(padre1.genes[i]);
    hijo2.genes.push_back(padre2.genes[i]);
  }
  // segunda parte: 1 a 2 y 2 a 1
  for(i=punto_cruce; i < lcrom; i++){
    hijo1.genes.push_back(padre2.genes[i]);
    hijo2.genes.push_back(padre1.genes[i]);
  }
  // se evalúan
  hijo1.aptitud = adaptacion(hijo1, lcrom, x_min, x_max);
  hijo2.aptitud = adaptacion(hijo2, lcrom, x_min, x_max);
}


/******************************************************************/
/*                                                                */
/* funcion: mutacion                                              */
/* Aplica el operador de mutacion a la poblacion                  */
/*                                                                */
/******************************************************************/
void mutacion(TPoblacion& pob, int tam_pob, int lcrom, double
prob_mut, double x_min, double x_max)
{
  bool mutado;
  int i,j;
  float prob;

  for (i=0; i < tam_pob; i++){
    mutado = false;
    for (j=0; j < lcrom; j++){
      // se genera un número aleatorio en [0 1)
      prob = (rand()/(RAND_MAX+1.0));
      // se mutan aquellos genes con prob < que prob_mut
      if ((prob < prob_mut) && (!pob[i].elite)){  //ELITISMO
        pob[i].genes[j] = !(bool)( pob[i].genes[j]);
        mutado = true;
      }
      if (mutado)
        pob[i].aptitud = adaptacion(pob[i], lcrom, x_min, x_max);
    }
  }
}
```

```
/*****************************************************************/
/*                                                               */
/* funcion: evaluacion                                           */
/* Calcula y actualiza as estadisticas de la poblacion           */
/*                                                               */
/*****************************************************************/

void evaluacion(TPoblacion& pob, int tam_pob, int& pos_mejor,
double& sumaptitud)
{
  float punt_acu = 0;     // puntación acumulada de los individuos
  float aptitud_mejor = 0;// mejor aptitud
  int i;
  sumaptitud = 0;  // suma de la aptitud

  for (i=0; i< tam_pob; i++){
    sumaptitud = sumaptitud + pob[i].aptitud;
    if (pob[i].aptitud > aptitud_mejor){
      pos_mejor = i;
      aptitud_mejor = pob[i].aptitud;
    }
  }
  for (i=0; i< tam_pob; i++){
    pob[i].puntuacion = pob[i].aptitud / sumaptitud;
    pob[i].punt_acu = pob[i].puntuacion + punt_acu;
    punt_acu = punt_acu + pob[i].puntuacion;
  }
}
```

Funciones.h para minimizar

Se introduce un nuevo campo en los individuos para almacenar la adaptación antes de transformarla para minimizar: aptitud_bruta. El resto del módulo no varía y por tanto no se incluye.

```
typedef struct{
          TGenes genes;       // cadena de bits (genotipo)
          double x;            // fenotipo
          double aptitud;      // función de adaptación
          doublé aptitud_bruta; // antes de transformarla
          double puntuacion; //punt. relativa:fitness/ sumfitness
          double punt_acu;     // puntuación acumulada para sorteos
          bool elite;          // indicativo de pertenecer a la élite
        }TIndividuo;
```

Funciones.cpp para minimizar

Representa la función:

- $f(x) = -|\,xsen(\sqrt{|\,x\,|})\,| : x \in [-250,250]$

que presenta un mínimo (-201.843) en $x = 203.814$.

El código es muy similar al presentado anteriormente. En el programa principal se incluye una llamada a la función *revisar_adap_mini*, para que realice la transformación de la adaptación bruta que proporciona la función de adaptación al valor de adaptación transformado para minimizar. Las funciones en las que se modifican individuos (*genena_indiv*, *cruce* y *mutación*) calculan la adaptación bruta, mientras que la función de selección trabaja con el valor de adaptación transformado.

```
/**********************************************************************
/*
/* Programacion evolutiva: proyecto 1                                */
/* version 2.0:  minimización                                        */
/* Modulo funciones.cpp                                              */
/*                                                                   */
/**********************************************************************

#include "funciones.h"


int main( int argc, char *argv[] )
{
  double x_min;            // límite inferior del intervalo
  double x_max;            // límite superior del intervalo
  double TOL;              // Tolerancia
  int tam_pob;             // tamaño de población
  int num_gen;             // número de generaciones
  double prob_cruce;       // probabilidad de cruce
  double prob_mut;         // probabilidad de mutación
  TPoblacion pob;          // población
  int lcrom;               // longitud de los individuos
  int generacion;          // contador de generaciones
  int pos_mejor;           // posición del mejor individuo
  double sumaptitud;       // suma de adaptaciones

  sscanf(argv[1], "%lf", &x_min ); // límite inferior del intervalo
                                   // en que se busca el óptimo
  sscanf(argv[2], "%lf", &x_max ); // límite superior del intervalo
                                   // en que se busca el óptimo
```

```
  sscanf(argv[3], "%lf", &TOL ); // tolerancia o exactitud
                                 // con que se busca el óptimo
  sscanf(argv[4], "%d", &tam_pob ); // tamaño de la población
  sscanf(argv[5], "%d", &num_gen ); // número de iteraciones
  sscanf(argv[6], "%lf", &prob_cruce ); // porcentaje de cruces
  sscanf(argv[7], "%lf", &prob_mut );  // porcentaje de mutaciones

  // inicializaciones
  lcrom = (int)ceil(log(1 + (x_max - x_min)/TOL) / log((double)2));

  // inicialización de números aleatorios
  time_t t;
  srand((unsigned) time(&t));


  // se genera la población inicial
  pob = poblacion_inicial(tam_pob, lcrom, x_min, x_max);
  // transformación de la adaptación para minimizar
  revisar_adap_mini(pob, tam_pob, lcrom, cmax, x_min, x_max);

  // se evalúa la población
  evaluacion(pob, tam_pob, pos_mejor, sumaptitud);

  // bucle de evolución
  for (generacion=0; generacion < num_gen; generacion++){
    seleccion(pob, tam_pob);
    reproduccion(pob, tam_pob, lcrom, prob_cruce, x_min, x_max);
    mutacion(pob, tam_pob, lcrom, prob_mut, x_min, x_max);
    revisar_adap_mini(pob, tam_pob, lcrom, cmax, x_min, x_max);
    evaluacion(pob, tam_pob, pos_mejor, sumaptitud);
  }
  cout << "Aptitud del mejor(" << pos_mejor << ") = ";
  cout <<  pob[pos_mejor].aptitud_bruta;
  cout << " en " << pob[pos_mejor].x << endl;
}

/*************************************************************/
/*                                                           */
/* funcion: revisar_adap_mini                                */
/* Revisa la aptitud de la pob para transforma mini en maxi */
/*                                                           */
/*************************************************************/

void revisar_adap_mini(TPoblacion& pob, int tam_pob, int lcrom,
               double& cmax, float x_min, float x_max)
{
  int i;
  cmax;

  cmax = pob[0].aptitud_bruta;
  for(i=1; i < tam_pob; i++){
    if (pob[i].aptitud > cmax)
```

```
      cmax = pob[i].aptitud_bruta;
    }
    cmax = cmax + cmax / 100;
    for(i=0; i < tam_pob; i++){
      pob[i].aptitud = cmax - pob[i].aptitud_bruta;
    }
}

/*****************************************************************/
/*                                                               */
/* funcion: poblacion_inicial                                    */
/* Genera la población inicial aleatoriamente                    */
/*                                                               */
/*****************************************************************/

TPoblacion poblacion_inicial(int tam_pob, int lcrom,
                             double x_min, double x_max)
{
  TPoblacion pob;
  TIndividuo indiv;
  int i;
  for(i=0; i < tam_pob; i++){
    indiv = genera_indiv(lcrom, x_min, x_max);
    pob.push_back(indiv);
  }
  return pob;
}

/*****************************************************************/
/*                                                               */
/* funcion: genera_indiv                                         */
/* Genera un individuo de la población inicial                   */
/*                                                               */
/*****************************************************************/

TIndividuo genera_indiv(int lcrom, double x_min, double x_max)
{
  int i;
  TIndividuo indiv;
  int gen;

  indiv.genes.clear();
  for(i=0; i < lcrom; i++){
    if (rand() < RAND_MAX / 2 )
      gen = 0;
    else
      gen = 1;
    indiv.genes.push_back(gen);
  }
  indiv.aptitud_bruta = adaptacion(indiv, lcrom, x_min, x_max);
  return indiv;
}
```

```
/*****************************************************************/
/*                                                               */
/* funcion: seleccion                                            */
/* Selecciona un unico individuo por el metodo de la ruleta      */
/*                                                               */
/*****************************************************************/

void seleccion(TPoblacion& pob, int tam_pob)
{
  int* sel_super;                   // seleccionados para sobrevivir
  float prob,f;                     // probabilidad de selección
  int pos_super;                    // posición del superviviente
  TPoblacion pob_aux;               // población auxiliar
  int i,j;
  int TAM_ELITE = pob.size()* 2 /100;
  int* sel_elite;

  sel_super = new int[tam_pob];
  sel_elite = new int[TAM_ELITE];
  //------------------------//
  // se selecciona la élite
  //------------------------//
  for (i=0; i < TAM_ELITE; i++){
    sel_elite[i] = 0;
  }
  for (i=0; i < tam_pob; i++){
    pob[i].elite = false;
    j = 0;
    while ( (j < TAM_ELITE) &&
        (pob[sel_elite[j]].aptitud >= pob[i].aptitud)){
      j++;
    }
    if (j < TAM_ELITE){
     sel_elite[j] = i;
    }
  }
  for (i=0; i < TAM_ELITE; i++){
    pob[sel_elite[i]].elite = true;
  }

  // se seleccionan tam_pob individuos para reproducirse
  // se generan números aleatorios entre 0 y 1,
  // seleccionan individuos de acuerdo con su puntuación acumulada

  for (i=0; i < tam_pob; i++){
    if (pob[i].elite == true)  //ELITISMO
      sel_super[i] = i;
    else{
      f = rand();
      prob = (f/(RAND_MAX+1.0));
      pos_super = 0;
```

```
      while ((prob>pob[pos_super].punt_acu)&&(pos_super < tam_pob))
        pos_super++;
      if (pos_super < tam_pob)
        sel_super[i] = pos_super;
      else
        sel_super[i] = pos_super-1;
    }
  }
  // se genera la población intermedia
  for (i=0; i < tam_pob; i++){
    pob_aux.push_back(pob[sel_super[i]]);
  }
  pob.clear();
  for (i=0; i < tam_pob; i++){
    pob.push_back(pob_aux[i]);
  }
  delete [] sel_super;
  delete [] sel_elite;
}

/*****************************************************************/
/*                                                               */
/* funcion: adaptacion                                           */
/* Evalua la calidad de un individuo                             */
/*                                                               */
/*****************************************************************/

double adaptacion(TIndividuo& individuo, int lcrom,
                  double x_min, double x_max)
{
  double x; // fenotipo
  double f; // valor de la función a optimizar

  x = decod(individuo.genes, lcrom, x_min, x_max);

  individuo.x = x;

  // AQUÍ se especifica la función a optimizar
  f = -(fabs(x * sin(sqrt(fabs(x)))));

  return f;
}


/*****************************************************************/
/*                                                               */
/* funcion: decod                                                */
/* Decodifica el genotipo de un individuo                        */
/*                                                               */
/*****************************************************************/

double decod(TGenes genes, int lcrom, double x_min, double x_max)
```

```
{
  double x;

  x=(float)bin_dec(genes,lcrom)/(pow((double)2,(double)lcrom) - 1);
  x = x_min + (x_max - x_min) * x;
  return x;
}


/*****************************************************************/
/*                                                               */
/* funcion: bin_dec                                              */
/* Convierte de binario a decimal                                */
/*                                                               */
/*****************************************************************/
int bin_dec(TGenes genes, int lcrom)
{
  int i, d=0, pot=1;

  for(i=0; i < lcrom; i++){
    d = d + pot*genes[i];
    pot = pot * 2;
  }
  return d;
}


/*****************************************************************/
/*                                                               */
/* funcion: reproduccion                                         */
/* Selecciona los individuos a reproducirse y aplica el          */
/* operador de cruce                                             */
/*                                                               */
/*****************************************************************/
void reproduccion(TPoblacion& pob, int tam_pob, int lcrom,
           double prob_cruce, double x_min, double x_max)
{
  Int* sel_cruce;        // seleccionados para reproducirse
  int num_sel_cruce=0; //num de individuos seleccionados para cruzar
  double prob;
  int punto_cruce;
  TIndividuo hijo1, hijo2;
  int i;

  sel_cruce = new int[tam_pob];
  // se eligen los individuos a cruzar
  for (i=0; i < tam_pob; i++){
    // se generan tam_pob números aleatorios a_i en [0 1)
    prob = (rand()/(RAND_MAX+1.0));
    // se eligen para el cruce los individuos de
    // las posiciones i con a_i < prob_cruce
    if (prob < prob_cruce){
```

```
        sel_cruce[num_sel_cruce] = i;
        num_sel_cruce++;
      }
  }
  // el número de selecionados se hace par
  if ((num_sel_cruce % 2) == 1)
    num_sel_cruce--;

  // se cruzan los individuos elegidos en un punto al azar
  punto_cruce = (int)(rand() * (lcrom+1) / (RAND_MAX + 1.0)) + 1;
  for (i=0; i < num_sel_cruce; i=i+2){
    cruce(pob[sel_cruce[i]], pob[sel_cruce[i+1]], hijo1, hijo2,
          lcrom, punto_cruce, x_min, x_max);
    // los nuevos individuos sustituyen a sus progenitores
    // respetando la élite
    if ((hijo1.aptitud<adaptacion(pob[sel_cruce[i]],lcrom,x_min,
x_max)) || (!pob[sel_cruce[i]].elite)){ //ELITISMO
      pob[sel_cruce[i]] = hijo1;
    }
    if ((hijo2.aptitud <
            adaptacion(pob[sel_cruce[i+1]], lcrom, x_min, x_max))
         || (!pob[sel_cruce[i+1]].elite)){
      pob[sel_cruce[i+1]] = hijo2;
    }
  }
  delete [] sel_cruce;
}

/**************************************************************/
/*                                                            */
/* funcion: cruce                                             */
/* Aplica el operador de cruce                                */
/*                                                            */
/**************************************************************/

void cruce(TIndividuo padre1, TIndividuo padre2, TIndividuo& hijo1,
TIndividuo& hijo2,
           int lcrom, int punto_cruce, double x_min, double x_max)
{
  int i;
  hijo1.genes.clear();
  hijo2.genes.clear();

  // primera parte del intercambio: 1 a 1 y 2 a 2
  for(i=0; i < punto_cruce; i++){
    hijo1.genes.push_back(padre1.genes[i]);
    hijo2.genes.push_back(padre2.genes[i]);
  }
  // segunda parte: 1 a 2 y 2 a 1
  for(i=punto_cruce; i < lcrom; i++){
    hijo1.genes.push_back(padre2.genes[i]);
    hijo2.genes.push_back(padre1.genes[i]);
```

```
    }
    // se evalúan
    hijo1.aptitud_bruta = adaptacion(hijo1, lcrom, x_min, x_max);
    hijo2.aptitud_bruta = adaptacion(hijo2, lcrom, x_min, x_max);
  }

/*************************************************************/
/*                                                           */
/* funcion: mutacion                                         */
/* Aplica el operador de mutacion a la población             */
/*                                                           */
/*************************************************************/

void mutacion(TPoblacion& pob, int tam_pob, int lcrom, double
prob_mut, double x_min, double x_max)
{
  bool mutado;
  int i,j;
  float prob;

  for (i=0; i < tam_pob; i++){
    mutado = false;
    for (j=0; j < lcrom; j++){
      // se genera un número aleatorio en [0 1)
      prob = (rand()/(RAND_MAX+1.0));
      // se mutan aquellos genes con prob < que prob_mut
      if ((prob < prob_mut) && (!pob[i].elite)){  //ELITISMO
        pob[i].genes[j] = !(bool)( pob[i].genes[j]);
        mutado = true;
      }
      if (mutado)
        pob[i].aptitud_bruta = adaptacion(pob[i], lcrom, x_min,
x_max);
    }
  }
}
/*************************************************************/
/*                                                           */
/* funcion: evaluacion                                       */
/* Calcula y actualiza as estadisticas de la población       */
/*                                                           */
/*************************************************************/

void evaluacion(TPoblacion& pob, int tam_pob, int& pos_mejor,
double& sumaptitud)
{
  float punt_acu = 0;     // puntacion acumulada de los individuos
  float aptitud_mejor = 0;   // mejor aptitud
  int i;
  sumaptitud = 0;  // suma de la aptitud
  for (i=0; i< tam_pob; i++){
    sumaptitud = sumaptitud + pob[i].aptitud;
```

```
    if (pob[i].aptitud > aptitud_mejor){
      pos_mejor = i;
      aptitud_mejor = pob[i].aptitud;
    }
  }
  for (i=0; i< tam_pob; i++){
    pob[i].puntuacion = pob[i].aptitud / sumaptitud;
    pob[i].punt_acu = pob[i].puntuacion + punt_acu;
    punt_acu = punt_acu + pob[i].puntuacion;
  }
}
```

Funciones.h para varias variables

Necesitamos representar ahora los genes correspondientes a cada una de las variables involucradas. Extendemos por lo tanto la representación del individuo:

```
typedef vector < bool > TGenes;      // cadena de bits (genotipo)
typedef vector< TGenes > Tvec_genes;// vector de genotipos
                                     // de cada variable
typedef vector< float > Tvec_var;    // vector de fenotipos
                                     // de cada variable
typedef vector< int > Tvec_int;      // vector para la longitudes
                                     // de los genes
typedef struct{
          Tvec_genes vec_genes;
          Tvec_var vec_x;           // fenotipo
          double aptitud_bruta; // función de evaluación
          double aptitud;     // aptitud transformada
          double puntuacion; // punt.relativa : fitness/sumfitness
          double punt_acu;    // puntuación acumulada para sorteos
        }TIndividuo;
```

Funciones.cpp para varias variables

Representa la función:

- $f(x_i \mid i=1..n) = \sum_{i=1}^{n} -x_i sen(\sqrt{|x_i|}) : x_i \in [0,100]$

que tiene un valor de $-n * 63.63498$ y se encuentra en $x_i = 65.54785$.

Las funciones se modifican sólo para aplicarse a todas las cadenas de genes incluidas en la representación. Incluimos aquí sólo las funciones que se ven afectadas:

```
/*****************************************************************/
/*                                                               */
/* funcion: genera_indiv                                         */
/* Genera un individuo de la población inicial                   */
/*                                                               */
/*****************************************************************/

TIndividuo genera_indiv(Tvac_int lcrom, Tvec_var x_min, Tvec_var
x_max)
{
  int i,j;
  TIndividuo indiv;
  TGenes genes;
  int gen;
  float f;

  indiv.vec_genes.clear();
  /*---------------------------------------------*/
  /* se generan los genotipos para cada variable */
  /*---------------------------------------------*/
  for(i=0; i< NUM_VAR; i++){
    genes.clear();
    for(j=0; j < lcrom[i]; j++){
      f = rand();
      if (f < RAND_MAX / 2 )
        gen = 0;
      else
        gen = 1;
      genes.push_back(gen);
    }
    indiv.vec_genes.push_back(genes);
  }
  indiv.aptitud_bruta = adaptacion(indiv, lcrom, x_min, x_max);
  return indiv;
}
/*****************************************************************/
/*                                                               */
/* funcion: adaptacion                                           */
/* Evalua la calidad de un individuo                             */
/*                                                               */
/*****************************************************************/

float adaptacion(TIndividuo& individuo, Tvec_int lcrom,
                 Tvec_var x_min, Tvec_var x_max)
{
  int i;
  Tvec_var x; // vector de fenotipos
```

```
  float f; // valor de la funcion a optimizar
  x.clear();
  individuo.vec_x.clear();
  for(i=0; i < NUM_VAR; i++){
    individuo.vec_x.push_back(
      decod(individuo.vec_genes[i], lcrom[i], x_min[i], x_max[i]));
  }
  x = individuo.vec_x;

  f = 0;
  for(i=0; i < NUM_VAR; i++){
    f = f - x[i] * sin(sqrt(fabs(x[i])));
  }
  return f;
}
/***************************************************************/
/*                                                             */
/* funcion: cruce                                              */
/* Aplica el operador de cruce                                 */
/*                                                             */
/***************************************************************/

void cruce(TIndividuo padre1, TIndividuo padre2, TIndividuo& hijo1,
       TIndividuo& hijo2,Tvec_int lcrom, Tvec_int punto_cruce,
       Tvec_var x_min, Tvec_var x_max)
{
  int i,j;
  TGenes genes_hijo1, genes_hijo2;

  hijo1.vec_genes.clear();
  hijo2.vec_genes.clear();

  for(j=0; j < NUM_VAR; j++){
    // primera parte del intercambio: 1 a 1 y 2 a 2
    genes_hijo1.clear();
    genes_hijo2.clear();
    for(i=0; i < punto_cruce[j]; i++){
      genes_hijo1.push_back(padre1.vec_genes[j][i]);
      genes_hijo2.push_back(padre2.vec_genes[j][i]);
    }
    // segunda parte: 1 a 2 y 2 a 1
    for(i=punto_cruce[j]; i < lcrom[j]; i++){
      genes_hijo1.push_back(padre2.vec_genes[j][i]);
      genes_hijo2.push_back(padre1.vec_genes[j][i]);
    }
    hijo1.vec_genes.push_back(genes_hijo1);
    hijo2.vec_genes.push_back(genes_hijo2);
  }
  // se evalúan
  hijo1.aptitud_bruta = adaptacion(hijo1, lcrom, x_min, x_max);
  hijo2.aptitud_bruta = adaptacion(hijo2, lcrom, x_min, x_max);
}
```

```
/*****************************************************************/
/*                                                               */
/* funcion: mutacion                                             */
/* Aplica el operador de mutacion a la población                 */
/*                                                               */
/*****************************************************************/
void mutacion(TPoblacion& pob, int tam_pob, Tvec_int lcrom,
              float prob_mut, Tvec_var x_min, Tvec_var x_max )
{
  bool mutado;
  int i,j,k;
  float prob,f;

  for (i=0; i < tam_pob; i++){
    mutado = false;
    for(k=0; k <NUM_VAR; k++){
      for (j=0; j < lcrom[k]; j++){
        // se genera un número aleatorio en [0 1)
        f = rand();
        prob = (f/(RAND_MAX+1.0));
        // se mutan aquellos genes con prob < que prob_mut
        if ((prob < prob_mut) && (!pob[i].elite)){
          pob[i].vec_genes[k][j] = !(bool)( pob[i].vec_genes[k][j]);
          mutado = true;
        }
      }
    }
    if (mutado)
     pob[i].aptitud_bruta = adaptacion(pob[i], lcrom, x_min, x_max);
  }
}
```

BIBLIOGRAFÍA

[ALB01] Alba, E., Troya, J.M. *Analyzing synchronous and asynchronous parallel distributed genetic algorithms.* Future Generation Comp. Syst., 17(4), pág. 451-465, (2001).

[ALB02] Alba, E., Tomassini, M. *Parallelism and evolutionary algorithms*. IEEE Trans. Evolutionary Computation, 6(5), pág. 443-462, (2002).

[ANC03] Anchor, K.P., Zydallis, J.B., Gunsch, G.H., Lamont, G.B. *Different Multi-objective Evolutionary Programming Approaches for Detecting Computer Network Attacks.* Proc. Evolutionary Multi-Criterion Optimization. Second International Conference, EMO 2003, Springer. Lecture Notes in Computer Science. Volume 2632, pág. 707-721, (2003).

[APP06] Applegate, D. L.; Bixby, R. M.; Chvátal, V. & Cook. *The Traveling Salesman Problem*. Princeton Series in applied Mathematics, (2006).

[BAC91] Bäck, T., Hoffmeister, F. Schwefel, H.P.. *A survey of evolution strategies.* Proc. of the Fourth International Conference on Genetic Algorithms, Morgan Kaufman, pág. 2-9, (1991).

[BAC94] Bäck, T. *Selective pressure in evolutionary algorithms: A characterization of selection mechanisms.* Proc. of the First IEEE Conf. on Evolutionary Computation, IEEE Press, pág. 57-62, (1994).

[BAC96A] Bäck, T.. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, (1996).

[BAC96B] Back, T., Schutz, M.. *Intelligent mutation rate control in canonical genetic algorithms.* Proc. International Syposium on Methodologies for Intelligent Systems, pág. 158-167, (1996).

[BAR00] Baraglia, R., Hidalgo, J.I and Perego, R. *A hybrid heuristic for the travelling salesman problem*. IEEE Trans. Evolutionary Computation 5(6): 613-622, (2001).

[BAR04] Barbosa, V.C., Assis,C.A.G., and do Nascimento, J.O. *Two novel evolutionary formulations of the graph coloring problem.* J. Comb. Optim., 8(1), pág. 41-63,(2004).

[BIA93] Bianchini, R., Brown, C. *Parallel genetic algorithms on distributed-memory architectures*. Proc. of the sixth conference of the North American Transputer Users Group on Transputer research and applications 6, IOS Press, pág. 67-82, (1993).

[CAN00] Cantu-Paz, E. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, (2000).

[CAN97A] Cantú-Paz, E. *A survey of parallel genetic algorithms.* Technical report, Illinois Genetic Algoritms Laboratory, IlliGAL Report No. 97003, (1997).

[CAN97B] Cantu-Paz, E., Goldberg, D.E. *Modeling idealized bounding cases of parallel genetic algorithms.* Genetic Programming: Proc. of the Second Annual Conference, CA: Morgan Kaufmann, pág. 353-361, (1997).

[CAN97C] Cantu-Paz, E., Goldberg, D.E. *Predicting speedups of idealized bounding cases of parallel genetic algorithms.* Proc. of the Seventh International Conference on Genetic Algorithms, CA: Morgan Kaufmann, pág. 113-120, (1997).

[CHE99] Cheng, V., Crawford, L., Menon, P. *Air traffic control using genetic search techniques*. Proc. of the IEEE International Conference on Control Applications, IEEE Computer Society, pág. 249-254, (1999).

[COE01] Coello Coello, C.A. *A Short Tutorial on Evolutionary Multiobjective Optimization*. First International Conference on Evolutionary Multi-Criterion Optimization, Springer-Verlag. Lecture Notes in Computer Science No. 1993, pág. 21-40, (2001).

[COE02] Coello Coello, C.A., Van Veldhuizen, D.A., Lamont, G.B.. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic Publishers, (2002).

[COL82] Colorni, A., Dorigo, M., y Maniezzo, V., *Genetic Algorithms: A new approach to the time-table problem*. Lecture Notes in Computer Science - NATO ASI Series, Vol.F 82, Combinatorial Optimization, Springer-Verlag, 235–239, (1982).

[CRA92] Crawford, K.D. *Solving the n-queens problem using genetic algorithms*. SAC '92: Proc. of the 1992 ACM/SIGAPP symposium on Applied computing, ACM, pág. 1039-1047, (1992).

[CRES05] Crescenzi, P., Kann, V., Halldórsson, M., Karpinski, M., Woeginger, G., *A Compendium of NP Optimization Problems*. KTH NADA, Stockholm, (2008).

[CRO05] Croitoru, C., Gheorghies, O., Gheorghies, A. *An ordering-based genetic approach to graph coloring*. Technical Report TR 05-03, "Al.I.Cuza" University of Iasi, Faculty of Computer Science, URL:http://www.infoiasi.ro/tr/tr.pl.cgi, (2005).

[CRO58] Croes, G.A., *A method for solving travelling salesman problem*. Operations Res. 6, pág. 791-812, (1958).

[DAR1859] Darwin, C.. *On the Origin of Species*. John Murray, (1859).

[DAV85] Davis, L., *Applying Adaptive Algorithms to Epistatic Domains*, Proceedings of the International Joint Conference on Artificial Intelligence, pág. 162-164. (1985).

[DAV91] Davis, L. *Handbook on genetic algorithm*. Van Nostral Reinhol New York, (1991).

[DAW76] Dawkins, R. *The Selfish Gene*. Oxford University Press, Oxford, (1976).

[DEB01] Deb, K., Kalyanmoy, D. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, (2001).

[DEB02] Deb, K., Anand, A., Joshi, D. *A computationally efficient evolutionary algorithm for real-parameter evolution*. Evolutionary Computation vol. 10 (4), pág. 371-395, (2002).

[DEB02] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T. *A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE transactions on Evolutionary Computations, 6(2):* pág. 182-197, (2002).

[DEB95] Deb, K., & Agarwal, R. (1995). *Simulated binary crossover for continuous search space*. Complex Systems, 9, 115–148, (1995).

[DOR96] Dorigo, M., Maniezzo, V., Colorni, A. *The Ant System: Optimization by a colony of cooperating agents*. IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics, 26(1): pág. 29-41, (1996).

[DOR97] Dorigo, M., Gambardella, L.M.. *Ant colony system: A cooperative learning approach to the traveling salesman problem*. IEEE Transactions on Evolutionary Computation, 1(1): pág. 53-66, (1997).

[EIB98] Eiben, A.E., van der Hauw, J.K., van Hemert, J.I. *Graph coloring with adaptive evolutionary algorithms*. J. Heuristics, 4(1): pág. 25-46, (1998).

[ESH89] Eshelman, L.J., Caruana, R., Schaffer, J.D. *Biases in the crossover landscape*. Proc. of the 3rd International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., pág. 10-19, (1989).

[ESH93] Eshelman L.J., Schaffer J.D. *Real-Coded Genetic Algorithms and Interval-Schemata*. Foundations of Genetic Algorithms, 2, pág. 187-202. (1993).

[FAL96] Falkenauer, E. *A hybrid grouping genetic algorithm for bin packing*. J. Heuristics, 2(1): pág. 5-30, (1996).

[FER02] Fernández de Vega, F., Galeano Gil, G., Gómez Pulido, J.A.. *Comparing synchronous and asynchronous parallel and distributed genetic programming models.* EuroGP '02: Proc. of the 5th European Conference on Genetic Programming, Springer-Verlag, pág. 326-336, (2002).

[FOG89] Fogarty T.C. *Learning new rules and adapting old ones with the genetic algorithm.* Proc. of the Fourth International Conference on Applications of Artificial Intelligence in Engineering, Springer, pág. 275-290. (1989).

[FON93] Fonseca, C.M., Fleming, P.J.. *Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization*. Genetic Algorithms: Proc. of the Fifth International Conference, Morgan Kaufmann, pág. 416-423, (1993).

[FON95] Fonseca, C.M., Fleming, P.J. *An overview of evolutionary algorithms in multiobjective optimization*. Evolutionary Computation, 3(1): pág. 1-16, (1995).

[FOR85] Forrest, S. *Documentation for prisoners dilemma and norms programs that use the genetic algorithim.* Technical report, University of Michigan, Ann Arbor, MI, (1985).

[GAL99] Galinier, P., Hao, J.K.. *Hybrid evolutionary algorithms for graph coloring*. J. Comb. Optim., 3(4): pág. 379-397, (1999).

[GAR79] Garey, M., and Johnson, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, (1979).

[GLA03] Glass, C.A., Prügel-Bennett, A. *Genetic algorithm for graph coloring: Exploration of galinier and hao's algorithm*. J. Comb. Optim., 7(3): pág. 229-236, (2003).

[GOL85] Goldberg, D.E. y Lingle, R., *Alleles, Loci and the TSP problem*, pág. Proceedings of the 1st International Conference on Genetic Algorithms, pág.154-159, (1985).

[GOL89] Goldberg, D.E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesly, (1989).

[GOL91] Goldberg, D.E. *Real-coded genetic algorithms, virtual alphabets, and blocking.* Complex Systems, 5: pág. 139-167, (1991).

[GOL91] Goldberg, D.E., Deb, K. *A comparative analysis of selection schemes used in genetic algorithms.* In FGA1, pág. 69-93, (1991).

[GRE87] Grefenstette, J.J., *Incorporating Problem Specific Knowledge into Genetic Algorithms*, pág., 42-60, Davis, L. D. (Ed.), London: Pitman, (1987).

[GRU94] Gruau, F. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, France, (1994).

[HAN94] Hancock, P.J.B. *An empirical comparison of selection methods in evolutionary algorithms*. Evolutionary Computing, AISB Workshop, pág. 80-94, (1994).

[HER05] F. Herrera, M. Lozano (Eds.) *Special Issue on Real Coded Genetic Algorithms: Foundations, Models and Operators*. SoftComputing 9: 4, (2005).

[HER98] Herrera, F., Lozano, M., Verdegay, J.L. *Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis.* Artificial Intelligence Review, 12(4): pág. 265-319, (1998).

[HER99] Herrera, F., Lozano, M., Moraga, C. *Hierarchical distributed genetic algorithms*. International journal of intelligent systems, 14(11): pág. 1099-1121, (1999).

[HOL75] Holland, J.H. *Adaptation in Natural and Artificial Systems*. Ann Harbor, MI: Univ. of Michigan Press, (1975).

[HOM92] Homaifar, A., Turner, J., Ali, S. *The n-queens problem and genetic algorithms.* Proc. IEEE Southeast Conference, Volume 1, pág. 262-267, (1992).

[IIM03] Iima, H., Yakawa, T. *A new design of genetic algorithm for bin packing*. Proc. of the 2003 Congress on Evolutionary Computation CEC2003, IEEE Press, pág. 1044-1049, (2003).

[JAC05] Jackson, D. *Evolving Defence Strategies by Genetic Programming*. EuroGP05: European Conference on Genetic Programming. Springer, pág. 281-290, (2005).

[JON74] Johnson, D. S., *Approximation algorithms for combinatorial problems, J. Comput. System*. Sci. 9, pág. 256–278, (1974).

[JON75] De Jong, K.A. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA, (1975).

[JON92] De Jong, K.A., Spears, W.M. *A formal analysis of the role of multi-point crossover in genetic algorithms.* Ann. Math. Artif. Intell., 5(1): pág. 1-26, (1992).

[JON95] De Jong, K.A. *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, Ann Arbor. Dissertation Abstracts International 36(10), 5140B; UMI 76-9381, (1995).

[KON04] Konfrst, Z. *Parallel genetic algorithms: Advances, computing trends, applications and perspectives.* Proc. of IPDPS04, (2004).

[KON06] Konak, A., Coit, D.W., Smith, A.E. *Multi-objective optimization using genetic algorithms: A tutorial.* Reliability Engineering & System Safety, 91(9): pág. 992-1007, (2006).

[KOZ02] Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, (1992).

[KOZ90] Koza, J. R. *Genetically breeding populations of computer programs to solve problems in artificial intelligence*. Proceedings of the Second International Conference on Tools for AI. Herndon, Virginia, Los Alamitos, IEEE Computer Society Press, pág. 819-827, (1990).

[KOZ92] Koza, J. R. *Genetic Programming: On the Programming of Computers by means of Natural Selection*. MIT Press, Massachussets, (1992).

[KOZ94] Koza, J. R., *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[KRA05] Krasnogor, N., Smith, J. *A tutorial for competent memetic algorithms: model, taxonomy, and design issues*. IEEE Trans. Evolutionary Computation, 9(5): pág. 474-488, (2005).

[LAR01] Larrañaga. P., Lozano, J.A. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, (2001).

[LAR03] P. Larrañaga, J.A. Lozano, and H. Mühlenbein. *Algoritmos de estimación de distribuciones en problemas de optimización combinatoria*. Inteligencia Artificial, Revista Iberoamericana de IA, 7(19): pág. 149-168, (2003).

[LAR08] Jiménez Laredo, J.L., Castillo, P.A., Mora, A.M., Merelo Guervós, J.J. *Evolvable agents, a fine grained approach for distributed evolutionary computing: walking towards the peer-to-peer computing frontiers.* Soft Computing 12(12), pág. 1145-1156, (2008).

[MCG85] Mcgilvary Gillies, A. *Machine learning procedures for generating image domain feature detectors*. PhD thesis, Ann Arbor, MI, USA, (1985).

[MIC94] Michalewicz, Z. *Genetic algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 2nd edition, (1994).

[MIC96] Michalewicz, Z., *Genetic algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 3nd edition, (1996).

[MOR06] Moraglio, A., Togelius, J., Lucas, S. *Product Geometric Crossover for the Sudoku Puzzle Evolutionary Computation*. CEC 2006. IEEE Congress on Volume , Issue, pag:470 – 476. (2006).

[MOS03] Moscato, P., Cotta Porras, C. *Una introducción a los algoritmos memeticos.* Inteligencia Artificial, Revista Iberoamericana de IA, 7(19): pág. 131-148, (2003).

[MOS99] Moscato, P. *Memetic algorithms: a short introduction.* New ideas in optimization, McGraw-Hill Ltd., pág. 219-234, (1999).

[MUH96] Mühlenbein, H., Paass, G. From recombination of genes to the estimation of distributions i. binary parameters. PPSN IV: Proc. of the 4th International Conference on Parallel Problem Solving from Nature, Springer-Verlag, pág. 178-187, (1996).

[OLI87] Oliver, I.M., Smith, D.J., y Holland, J.R.C., A *study of permutation crossover operators on the travelling salesman problem*. Proceedings of the Second International Conference on Genetic Algorithms, pág. 224-230, (1987).

[PER96] Perez Serrada, A.. *Una introducción a la computación evolutiva.* Technical report, "ftp://ftp.de.uu.net/pub/research/softcomp/EC/EA/papers/intro-spanish.ps.gz", (1996).

[PRI05] Price, K., Storn, R.M., Lampinen, J.A. *Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series)*. Springer-Verlag , (2005).

[PUC04] Puchinger, J., Raidl, G.R., Koller, G.. *Solving a real-world glass cutting problem.* Proc. of EvoCOP, volume 3004 of Lecture Notes in Computer Science, pág. 165-176. Springer, (2004).

[RIZ07] Rizzoli,A.E.,Montemanni, R., Lucibello, E., Gambardella, L.M. *Ant colony optimization for real-world vehicle routing problems*. Swarm Intelligence, 1(2): pág. 135-151, (2007).

[ROS06] Rossi, F., van Beek, P., Walsh, T. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., (2006).

[ROT06] Rothlauf, F. *Representations for Genetic and Evolutionary Algorithms*. 2nd edition. Springer, Heidelberg New York, (2006).

[RYA98] Ryan, C., Collins, J.J., O Neill, M. *Grammatical evolution: Evolving programs for an arbitrary language.* Proc. of the First European Workshop on Genetic Programming, volume 1391, Springer-Verlag, pág. 83-95, (1998).

[SHA85] Schaffer, J.D. *Learning multiclass pattern discrimination.* Proc. of the International Conference on Genetic Algorithms and Their Applications, pág. 74-79, (1985).

[SHA87] Schaffer, J.D., Morishima, A. *An adaptive crossover distribution mechanism for genetic algorithms.* Proc. of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application, Lawrence Erlbaum Associates, Inc., pág. 36-40, (1987).

[SHI06] Watanabe, S., Sakakibara, K. *A multiobjectivization approach for vehicle routing problems.* Proc. of *EMO*, pág. 660-672, (2006).

[SPE91] Spears, W.M., De Jong, K.A. *On the virtues of parameterized uniform crossover.* Proc. of the Fourth International Conference on Genetic Algorithms, Morgan Kaufman, pág. 230-236, (1991).

[STO97] Storn, R., Price, K. *Differential evolution -- a simple and efficient heuristic for global optimization over continuous spaces*. J. of Global Optimization, 11(4): pág. 341-359, (1997).

[SYS89] Syswerda, G. *Uniform crossover in genetic algorithms*. Proc. of the Third International Conference on Genetic Algorithms(ICGA-80), Morgan Kaufmann, pág. 2-9 (1989).

[TSA99] Tsang. E. *A glimpse of constraint satisfaction*. Artif. Intell. Rev., 13(3): pág. 215-227, (1999).

[WAT03] Watanabe, S., Hiroyasu,T., Miki, M. Multi-objective Rectangular Packing Problem and Its Applications. Proc. of, EMO 2003, Lecture Notes in Computer Science. Volume 2632, Springe, pág. 565-577, (2003).

[WHI93] Whitley, L.D. *Cellular genetic algorithms*. Proceeding of *ICGA*, pág. 658, (1993).

[ZHO05] Zhou, Y., Han, R.P.S. *A genetic algorithm with elite crossover and dynastic change strategies*. ICNC *(3)*, pág. 269-278, (2005).

ÍNDICE ALFABÉTICO

8

A

B

C

N

O

P

R

S

T

V

Algoritmos Evolutivos

Un enfoque práctico

algoritmos evolutivos constituyen una técnica general de resolución de
lemas de búsqueda y optimización inspirada en la teoría de la evolución de las
ecies y la selección natural. Estos algoritmos permiten abordar problemas
plejos que surgen en las ingenierías y los campos científicos: problemas de
ificación de tareas, horarios, tráfico aéreo y ferroviario, búsqueda de caminos
mos, optimización de funciones, etc. Con este libro hemos querido aportar un
que práctico al estudio de los algoritmos evolutivos, que es fundamental para
carlos a problemas reales de cualquier disciplina del conocimiento. El libro tiene
partes: la primera, en la que se describen los algoritmos; y la segunda en la que
roponen numerosos proyectos y se resuelven empleando estas técnicas.

algoritmos evolutivos presentan una estructura general que puede aplicarse a
distintos problemas, facilitando así enormemente las tareas de diseño e
ementación. El único requisito de un usuario que desee aplicar esta técnica para
lver un problema concreto es saber programar en cualquier lenguaje de
ósito general en el que codificaría el algoritmo evolutivo. Sin embargo, para
ner buenos resultados con estos algoritmos es necesario conocerlos con
lle, ya que dentro del esquema general de un algoritmo evolutivo hay que elegir
iples componentes y parámetros, de los que va a depender la calidad del
ltado y la eficiencia del algoritmo. El conocimiento de la elección más adecuada
ada caso, que a menudo depende de detalles sutiles del problema considerado,
se consigue con la práctica. Esta idea nos ha llevado a proponer este libro, que
sideramos adecuado para cualquier ingeniero o licenciado con conocimientos
cos de programación.

w.alfaomega.com.mx



"Te acerca al conocimiento"
Alfaomega Grupo Editor